嵌入式技术与应用丛书

汽车电子 KEA 系列微控制器
——基于 ARM Cortex-M0+内核

王宜怀　李跃华　著

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

本书阐述飞思卡尔 2014 年开始推出的面向汽车电子 KEA 系列微控制器的应用方法。全书共 16 章，1～6 章囊括了学习一个新微控制器入门的基本知识要素及基本规范，7～12 章分别给出各个模块的程序设计方法，第 13 章给出时钟系统及其他模块，第 14 章给出基于实时操作系统 MQX-Lite 的编程方法，第 15 章给出一个汽车冷却风扇控制实例，第 16 章给出位带及位操作引擎等技术的进一步讨论。

本书提供了网上光盘，内含所有有关文档、工程框架、各章实例源程序及常用软件工具。网上光盘下载地址：http://sumcu.suda.edu.cn。

本书适用于从事汽车电子及嵌入式技术的研发人员、工程师、高校教师、研究生、高年级本科生使用，也可作为嵌入式技术培训材料。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

汽车电子 KEA 系列微控制器：基于 ARM Cortex-M0+内核 / 王宜怀，李跃华著. —北京：电子工业出版社，2015.8
（嵌入式技术与应用丛书）
ISBN 978-7-121-26686-7

I. ①汽… II. ①王… ②李… III. ①微控制器 IV. ①TP332.3

中国版本图书馆 CIP 数据核字（2015）第 164453 号

责任编辑：田宏峰
印　　刷：北京中新伟业印刷有限公司
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编　100036
开　　本：787×1 092　1/16　印张：23.25　字数：590 千字
版　　次：2015 年 8 月第 1 版
印　　次：2015 年 8 月第 1 次印刷
印　　数：3000 册　　定价：68.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

飞思卡尔于 2014 年开始推出的面向汽车电子 KEA 系列微控制器，该微控制器以 ARM Cortex-M0+为内核，内核频率 48 MHz，运行温度范围–40 ℃～125 ℃，内含温度传感器，具有位操作引擎及良好的 EMC 特性，将广泛地应用于汽车电子中的信息娱乐系统连接、停车辅助系统、车身中控、空调控制、车窗/天窗/车门控制、防盗装置、通用传感器节点等。整个 KEA 系列引脚兼容，还将与未来的 Kinetis Auto 系列的其他系列产品引脚兼容，并提供多个引脚和存储选择，使开发人员能够在不同芯片之间进行移植，最大限度地提高硬件、软件重用性并缩短投放市场的时间。相比于一般微控制器，KEA 系列微控制器具有更宽的温度范围及更高的可靠性。因其具有性能好、可靠性高、价格低等突出特点，将是汽车电子领域的重要成员，也能更好地应用于其他对可靠性要求较高的领域。

本书以构件化底层驱动为基础阐述 KEA 系列微控制器应用程序的设计方法。

主要特点

（1）按照由浅入深的原则，在简要给出基本软件、硬件要素的基础上，尽快给出符合嵌入式软件工程规范的工程框架、第一个 C 语言样例、第一个汇编样例及启动过程解析，以便快速规范入门。一些较深的内容，随后给出。

（2）对底层驱动进行构件化封装。书中对每个模块均给出了根据嵌入式软件工程基本原则并按照构件化封装要求，编制面向芯片级底层驱动程序，同时给出较为详细、规范的注释及对外接口，为实际应用提供底层构件，方便移植与复用，可以为读者进行实际项目开发节省大量时间。面向芯片级底层驱动程序为芯片的实际应用提供了基础，避免了面向板级驱动给不用用户带来的不便。

（3）设计合理的测试用例。书中所有源程序均经测试通过，并保留测试用例在本书的网上光盘中，避免了因例程的书写或固有错误给读者带来烦恼。这些测试用例，也为读者验证与理解带来方便。每个测试用例均为独立的工程，包含文档说明，为读者重复实验现象提供便利。书中附录还提供了 KDS 简明使用方法、写入器驱动与使用方法、部分工具软件。

（4）网上光盘提供了所有模块完整的底层驱动构件化封装程序与测试用例。需要使用 PC 的程序的测试用例，还提供了 PC 的 C#源程序。网上光盘的版本将会适时更新。

（5）提供硬件核心板、写入调试器等硬件，方便读者进行实践与应用。

主要内容

全书共 16 章，第 1 章给出汽车电子概述及 KEA 的基本特性；第 2 章给出 ARM Cortex-M0+的寄存器、指令系统及汇编语法；第 3 章给出 KEA 的存储地址映像、中断系统及最小硬件系统；第 4 章以 GPIO 为例给出工程框架、第一个 C 语言实例、第一个汇编语言实例及程序执行过程分析；第 5 章给出底层驱动构件设计规范；第 6 章给出串行通信及 printf 函数的融入方法。1～6

章囊括了学习一个新微控制器入门的基本知识要素及基本规范。第 7～12 章分别给出了 Systick、RTC、PWT、PIT、FTM（含输入捕捉、输出比较及 PWM）、Flash 在线编程、ADC、ACMP、SPI、I2C、CAN 等程序设计方法；第 13 章给出时钟系统及其他模块；第 14 章给出基于实时操作系统 MQX-Lite 的编程方法；第 15 章给出一个汽车冷却风扇控制实例；第 16 章给出位带及位操作引擎等技术的进一步讨论。

作者分工与致谢

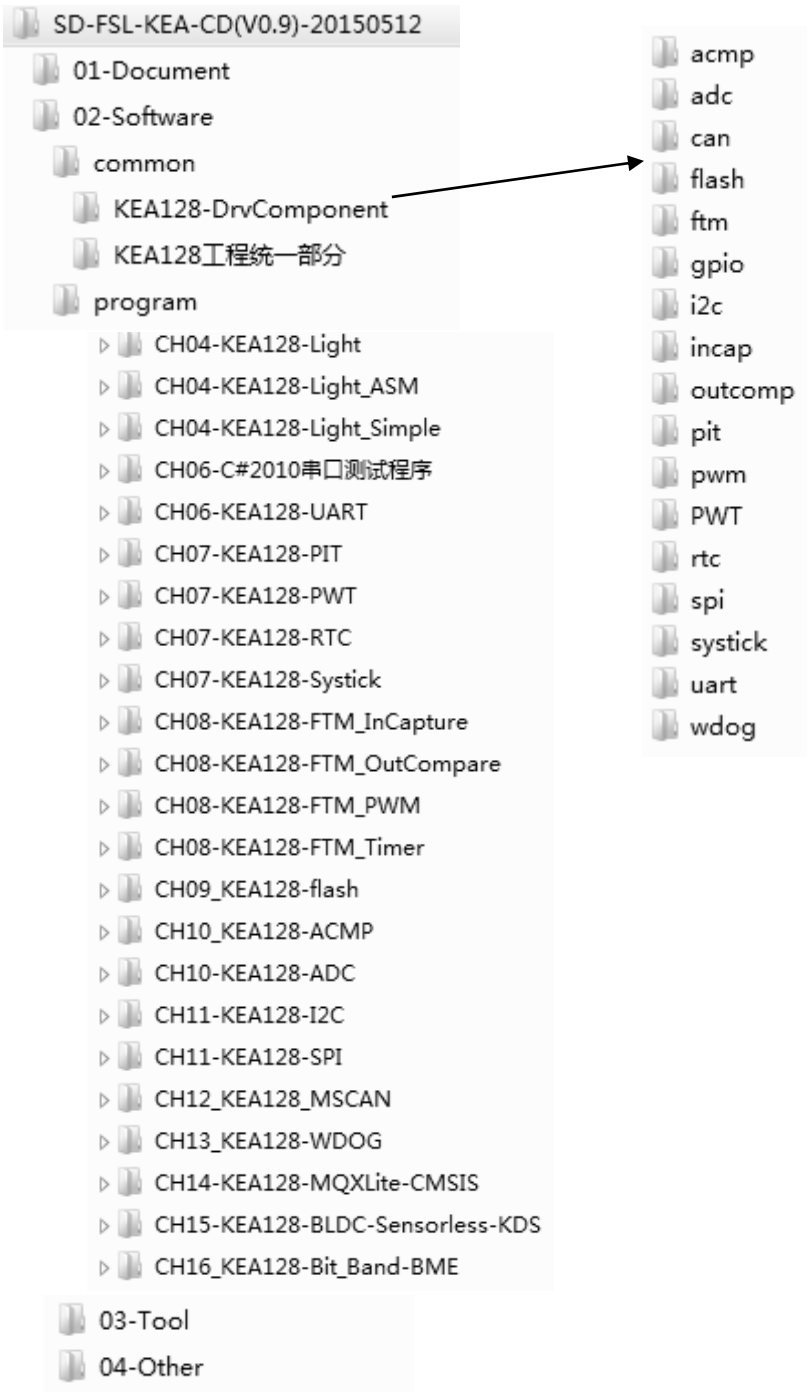
本书由王宜怀负责编制提纲和统稿工作，并撰写第 2～6 章。李跃华撰写 7～14 章及第 16 章部分内容。飞思卡尔李兴参与全书的策划，田茂昕撰写第 1 章，唐华标撰写第 15 章及第 16 章部分内容。研究生李会、王磊、陈瑞杰、范宁宁等协助书稿整理及程序调试工作，他们卓有成效的工作，使本书更加实用。飞思卡尔马莉女士一直关心支持苏州大学飞思卡尔嵌入式中心的建设，为本书的撰写提供了技术条件。飞思卡尔李越、陈馨宇等工程人员提供了技术支持。苏州华祥信息科技有限公司的蒋建辉、刘辉工程师提出了有益的建议。电子工业版社的田宏峰先生为本书的出版做了大量细致的工作。在此一并表示诚挚的谢意。

鉴于作者水平有限，书中难免存在不足和错误之处，恳望读者提出宝贵意见和建议，以便再版时改进。

王宜怀

2015 年 6 月

网上光盘资料文件夹结构



该网上光盘资料会不定期更新，下载路径：http://sumcu.suda.edu.cn→“教学与培训”→“KEA”→“SD-FSL-KEA-CD”。

目　　录

第 1 章　概述……1
1.1　汽车电子技术的基本概念……1
1.2　中国汽车电子发展概况……2
1.3　飞思卡尔在汽车电子市场中的地位……2
1.4　面向汽车电子的微控制器 KEA 系列 MCU 简介……3

第 2 章　ARM Cortex-M0+处理器……7
2.1　ARM Cortex-M0+处理器简介……7
2.1.1　ARM Cortex-M0+处理器特点与结构图……8
2.1.2　ARM Cortex-M0+处理器存储器映像……9
2.1.3　ARM Cortex-M0+处理器的寄存器……10
2.2　ARM Cortex-M0+处理器的指令系统……13
2.2.1　ARM Cortex-M0+指令简表与寻址方式……13
2.2.2　数据传送类指令……14
2.2.3　数据操作类指令……16
2.2.4　跳转控制类指令……19
2.2.5　其他指令……20
2.3　ARM Cortex-M0+指令集与机器码对应表……21
2.4　GNU 汇编语言的基本语法……23
2.4.1　汇编语言格式……23
2.4.2　伪指令……25

第 3 章　KEA128 存储映像、中断与硬件最小系统……29
3.1　KEA128 系列存储映像……29
3.1.1　Flash 区存储映像……30
3.1.2　片内 RAM 区存储映像……30
3.1.3　外设区存储映像……30
3.1.4　私有外设总线存储映像……31
3.1.5　系统保留段存储映像……31
3.2　KEA128 中断系统及 Cortex-M0+非内核模块中断编程结构……31
3.2.1　KEA128 的中断源……31
3.2.2　KEA128 中断向量表……32
3.2.3　Cortex-M0+非内核模块中断编程结构……36
3.3　KEA128 的引脚功能……38
3.3.1　硬件最小系统引脚……39
3.3.2　I/O 端口资源类引脚……39
3.4　KEA128 硬件最小系统原理图……40

3.4.1 电源及其滤波电路……40
3.4.2 复位电路及复位功能……40
3.4.3 SWD 接口电路……41
3.4.4 晶振电路……41
第 4 章 GPIO 及程序框架……43
4.1 KEA128 芯片 GPIO 驱动构件及使用方法……43
4.1.1 KEA128 芯片 GPIO 引脚……44
4.1.2 KEA128 芯片 GPIO 驱动构件头文件及使用方法……45
4.2 GPIO 驱动构件的制作方法……47
4.2.1 端口控制模块功能与编程结构……47
4.2.2 GPIO 模块的编程结构……50
4.2.3 GPIO 基本打通程序……51
4.2.4 GPIO 驱动构件封装要点分析……51
4.2.5 GPIO 驱动构件的实现……53
4.3 第一个 C 语言工程：控制小灯闪烁……59
4.4 工程框架与第一个 C 语言工程执行过程分析……63
4.4.1 工程框架……63
4.4.2 链接文件……64
4.4.3 机器码文件……65
4.4.4 其他相关文件功能简介……67
4.4.5 芯片上电启动执行过程……68
4.5 第一个汇编语言工程：控制小灯闪烁……72
4.5.1 汇编工程文件的组织……72
4.5.2 Light 构件汇编程序 light.s……73
4.5.3 Light 测试工程主程序及汇编工程执行过程……74
第 5 章 嵌入式硬件构件与底层驱动构件基本规范……77
5.1 嵌入式硬件构件……77
5.1.1 嵌入式硬件构件的概念……77
5.1.2 基于嵌入式硬件构件的电路原理图设计简明规则……78
5.2 嵌入式底层驱动构件的概念与层次模型……80
5.2.1 嵌入式底层驱动构件的概念……81
5.2.2 嵌入式硬件构件和软件构件的层次模型……81
5.3 底层驱动构件的封装规范……82
5.3.1 构件设计的基本思想与基本原则……82
5.3.2 编码风格基本规范……84
5.3.3 公共要素文件……87
5.3.4 头文件的设计规范……89
5.3.5 源程序文件的设计规范……90
5.4 硬件构件及底层软件构件的重用与移植方法……91

第 6 章　串行通信模块及第一个中断程序结构 ……95
6.1　KEA128 芯片 UART 驱动构件及使用方法 ……95
6.1.1　UART 的基础知识要素 ……95
6.1.2　UART 驱动构件封装要点分析 ……96
6.1.3　KEA128 芯片 UART 引脚 ……97
6.1.4　KEA128 芯片 UART 驱动构件头文件及使用方法 ……98
6.2　UART 接收中断程序实例 ……101
6.2.1　KEA128 的中断服务程序及其“注册” ……101
6.2.2　UART 接收中断程序实例 ……103
6.2.3　printf 的设置方法与使用 ……107
6.3　UART 驱动构件的制作方法 ……107
6.3.1　UART 模块编程结构 ……107
6.3.3　UART 驱动构件的实现 ……111
第 7 章　Systick、RTC、PWT 及 PIT ……117
7.1　ARM Cortex-M0+内核时钟（Systick） ……117
7.1.1　Systick 模块的编程结构 ……117
7.1.2　Systick 构件设计及测试实例 ……118
7.2　实时时钟模块（RTC） ……120
7.2.1　RTC 模块概述与编程要点 ……120
7.2.2　RTC 构件设计及测试实例 ……122
7.3　脉冲宽度定时器（PWT） ……125
7.3.1　PWT 模块概述与编程要点 ……125
7.3.2　PWT 构件设计及测试实例 ……127
7.4　周期性中断定时器（PIT） ……131
7.4.1　PIT 模块概述与编程要点 ……131
7.4.2　PIT 构件设计及测试实例 ……133
第 8 章　Flex 定时器 FTM ……137
8.1　FTM 基本知识 ……137
8.1.1　FTM 概述 ……137
8.1.2　FTM 技术要点 ……137
8.1.3　FTM 寄存器总览 ……138
8.2　FTM 基本定时的编程结构与测试实例 ……140
8.2.1　FTM 基本定时的编程结构 ……140
8.2.2　FTM 基本定时构件与测试实例 ……142
8.3　FTM 模块的脉宽调制（PWM）功能 ……147
8.3.1　脉宽调试器 PWM 基本工作原理 ……147
8.3.2　KEA128 的三种 PWM 模式 ……148
8.3.3　FTM 引脚复用 ……149
8.3.4　PWM 构件与测试实例 ……150
8.4　FTM 模块的输出比较功能 ……159

8.4.1 输出比较的基本知识 159
8.4.2 输出比较构件与测试实例 160
8.5 FTM 模块的输入捕捉功能 168
8.5.1 输入捕捉基本含义 168
8.5.2 输入捕捉中断构件与测试实例 169
第 9 章 Flash 在线编程 179
9.1 KEA128 芯片 Flash 驱动构件及使用方法 179
9.1.1 Flash 编程知识要素 179
9.1.2 KEA128 芯片 Flash 构件头文件及使用方法 180
9.2 Flash 驱动构件的制作方法 183
9.2.1 Flash 模块编程结构 183
9.2.2 Flash 驱动构件制作要点 186
9.3 Flash 驱动构件封装要点分析及实现 189
9.3.1 Flash 驱动构件封装要点 189
9.3.2 Flash 驱动构件的实现 190
9.4 Flash 模块的保护与加密 203
9.4.1 Flash 模块的保护 203
9.4.2 Flash 模块的安全 206
第 10 章 ADC 与 ACMP 模块 207
10.1 KEA128 芯片 ADC 驱动构件及使用方法 207
10.1.1 ADC 编程知识要素 207
10.1.2 KEA128 芯片的 ADC 引脚与通道号 208
10.1.3 KEA128 芯片 ADC 构件头文件及使用方法 209
10.2 ADC 构件的制作方法 210
10.2.1 ADC 转换模块编程结构 211
10.2.2 ADC 构件封装要点和函数分析 215
10.2.3 ADC 驱动构件的实现 215
10.3 KEA128 芯片 ACMP 驱动构件及使用方法 220
10.3.1 ACMP 编程知识要素 220
10.3.2 ACMP 引脚的标识 220
10.3.3 KEA128 芯片 ACMP 构件头文件及使用方法 220
10.4 ACMP 构件的制作方法 222
10.4.1 ACMP 模块编程结构 222
10.4.2 ACMP 构件封装要点和函数分析 224
10.4.3 ACMP 驱动构件的实现 224
第 11 章 SPI 与 I2C 模块 231
11.1 SPI 模块 231
11.1.1 SPI 编程知识要素 231
11.1.2 KEA128 芯片 SPI 引脚 232

11.1.3 KEA128 芯片 SPI 构件头文件及使用方法……233
11.1.4 SPI 主从机通信实例……236
11.1.5 SPI 构件的制作方法……239
11.2 I2C 模块……247
11.2.1 I2C 编程知识要素……247
11.2.2 KEA128 芯片 I2C 引脚的标识……249
11.2.3 KEA128 芯片 I2C 构件头文件及使用方法……250
11.2.4 I2C 主从机通信实例……254
11.2.5 I2C 构件的制作方法……257
第 12 章 KEA128 的 MSCAN 总线开发方法……269
12.1 CAN 总线通用知识……269
12.1.1 CAN 硬件系统的原理性电路……269
12.1.2 CAN 总线的有关基本概念……270
12.2 MSCAN 驱动构件及使用方法……271
12.2.1 KEA128 芯片 MSCAN 编程知识要素……272
12.2.2 MSCAN 构件头文件及使用方法……274
12.3 MSCAN 驱动构件制作方法……277
12.3.1 MSCAN 寄存器简介……277
12.3.2 MSCAN 构件封装要点分析……282
12.4 MSCAN 驱动构件的设计……283
12.4.1 MSCAN 初始化……283
12.4.2 MSCAN 发送数据包函数……286
12.4.3 MSCAN 接收数据包函数……288
第 13 章 系统时钟与其他功能模块……291
13.1 时钟系统……291
13.1.1 时钟系统概述……291
13.1.2 时钟模块概要与编程要点……293
13.1.3 时钟模块测试实例……293
13.2 复位模块……294
13.2.1 上电复位……295
13.2.2 系统复位源……295
13.2.3 调试复位……296
13.3 看门狗……297
13.3.1 功能描述……297
13.3.2 配置 WDOG……297
13.3.3 测试实例……298
13.4 电源模式与芯片配置……298
13.4.1 电源模式控制……298
13.4.2 低功耗下的模块操作……298
13.4.3 芯片配置模块……298

13.5 循环冗余检查、杂项控制模块与交叉开关……299
13.5.1 循环冗余检查……299
13.5.2 杂项控制模块……299
13.5.3 交叉开关……299
第 14 章 KEA128 在实时操作系统 MQX-Lite 下的应用……301
14.1 MQX-Lite 简介……301
14.2 MQX-Lite 编程知识要素……302
14.2.1 任务管理与调度……302
14.2.2 任务间同步与通信……303
14.2.3 中断处理机制……304
14.3 基于 MQX-Lite 的 KEA128 工程框架……306
14.4 KEA128 在 MQX-Lite 下的第一个样例工程……307
14.4.1 样例工程的功能……307
14.4.2 样例工程任务设计……307
14.4.3 样例工程的执行流程及运行结果……314
第 15 章 基于 KEA 的无刷直流电机的汽车应用……317
15.1 无刷直流电机在汽车上的应用现状和发展趋势……317
15.2 无刷直流电机在 KEA128-BLDCRD 板上的操作指南……317
15.2.1 软硬件的准备……317
15.2.2 操作流程……318
15.3 无刷直流电机驱动的基本原理及无传感器控制……319
15.3.1 换向控制……320
15.3.2 转速转矩控制……321
15.3.3 互补型极性 PWM 调制技术……321
15.3.4 基于反电动过零的位置估计……322
15.3.5 无传感器 BLDC 的启动……326
15.4 基于 KEA 的车用无刷直流电机应用实例——车用冷却风扇……327
15.4.1 方案简介……327
15.4.2 硬件结构……327
15.4.3 电气指标……329
15.4.4 软件实现……329
15.4.5 开发中的相关工具……331
15.4.6 方案总结……334
15.5 关于车用 BLDC 的展望……334
第 16 章 有关问题的进一步讨论……335
16.1 位带技术及应用方法……335
16.1.1 位带别名区概述……335
16.1.2 位带别名区的应用机制解析……335
16.1.3 位带别名区使用注意点……337

16.2 位操作引擎技术及应用方法 ····· 338
16.2.1 位操作引擎概述 ····· 338
16.2.2 位操作引擎的应用机制解析 ····· 339
16.2.3 位操作引擎对 GPIO 部分的使用说明 ····· 341
16.2.4 位操作引擎使用注意点 ····· 342
16.2.5 测试实例 ····· 342
16.3 EMC 问题的探讨 ····· 342
16.3.1 简介 ····· 342
16.3.2 硬件设计 ····· 342
16.3.3 软件设计 ····· 344
16.4 基于 CAN 接口及 Bootloader 的程序更新方法 ····· 345
16.4.1 概述 ····· 345
16.4.2 操作指南 ····· 345
16.4.3 地址空间分布 ····· 347
16.4.4 CAN Bootloader 软件流程 ····· 347
16.5 AUTOSAR 简介 ····· 348
16.5.1 概述 ····· 348
16.5.2 AUTOSAR 软件架构 ····· 348
16.5.3 飞思卡尔与 AUTOSAR ····· 349
16.5.4 AUTOSAR 相关问题 ····· 349
附录 A SKEAZ128MLK 引脚功能分配 ····· 350
附录 B KEA128 最小系统 ····· 352
附录 C printf 格式化输出 ····· 353
参考文献 ····· 355

第 1 章

概　　述

本章导读：鉴于本书阐述面向汽车电子的微控制器，作为全书导引，本章首先简要阐述汽车电子技术的基本概念、我国汽车电子发展概况及飞思卡尔在汽车电子市场中地位。随后给出飞思卡尔面向汽车电子的微控制器 KEA 系列型号标识与基本特点。KEA 系列微控制器与一般工业级微控制器主要区别是提高了运行温度等级及电磁兼容等级等，当然也可用于一般工业控制领域。

本章参考资料：本章 1.4 节的 KEA 系列 MCU 的型号标识总结自《KEA128 数据手册》的第 2 章，KEA 系列 MCU 的基本特点及简明资源参考自《KEA 简介》。有关简称含义见本书参考文献，下同。

1.1　汽车电子技术的基本概念

汽车电子是车体汽车电子控制装置和车载汽车电子控制装置的总称，其最重要的作用是提高汽车的安全性、舒适性、经济性和娱乐性。近年来，随着信息技术的快速发展，以及汽车制造业水平的不断提升，汽车电子技术得到了广泛应用和创新，越来越多的电子控制单元（Electronic Control Unit，ECU）被集成在汽车中。电子设备在汽车总成本中所占的比例也随之攀升，据统计，1970 年全球汽车电子设备的成本占比为 2%，1980 年为 5%，1990 年为 15%，2000 年为 20%，2012 年为 25%；预计到 2015 年将升至 50%。这些汽车电子产品的加入，一方面对提高汽车的动力性、燃油经济性、安全性，以及降低汽车排放污染起到了非常关键的作用，另一方面提供了如仪表显示、信息娱乐、通信互连等丰富的人车交互系统及智能化的车身控制模块，大大增强了整车的驾驶舒适性。专家指出，近 10 年来汽车产业 70%的创新来源于汽车电子技术及其产品的开发应用，汽车电子技术的应用水平已成为衡量汽车档次水平的主要标志，其应用程度的提高是汽车生产企业提高市场竞争力的重要手段。

汽车电子化被认为汽车技术发展进程中的一次革命，是用来开发新车型、改进汽车性能最重要的技术措施。按照应用方向，汽车电子技术可划分成四类：第一类是车身电子，如车身控制器（BCM）、车内网关（Gateway），以及对车灯、车窗、车门等部件进行控制的车身节点；第二类是汽车动力总成，指对发动机、变速箱、驱动电机的控制；第三类是汽车底盘与安全，如对底盘、悬架、制动系统、电子助力转向（Electric Power Steering，EPS）等的控制，以及近期发展势头强劲的高级驾驶员辅助系统（Advanced Driver Assistance Systems，ADAS）；第四类是驾驶员信息系统（Driver Information System，DIS），包括仪表显示、信息娱乐、GPS 导航等。

1.2 中国汽车电子发展概况

自 2001 年正式加入 WTO 以来，中国巨大的市场潜力吸引了全球众多知名汽车厂商的目光，汽车产业因此得到了飞速发展。短短几年，中国一举成为世界最大的汽车生产国和消费国。据中汽协公布的统计数据显示，2013 年，中国汽车产销双双超过 2000 万辆，再创全球产销最高记录，其中产量 2211.68 万辆，销量 2198.41 万辆，同比增长 14.76%和 13.87%。预计未来较长的一段时间，中国都将保持汽车产销大国地位。

汽车产业的迅猛发展为汽车电子及相关半导体提供了广阔的市场前景。根据 Strategy Analytics 的最新数据显示，2013 年中国汽车半导体（包括 MCU、Sensor 以及 MOSFET、IGBT、三极管、二极管等分立器件，下同）的总需求量为 120.87 亿颗，较 2012 年增长 26.2%，其中 MCU 需求量为 4.34 亿颗，平均每辆汽车大约包含 19 颗 MCU；2013 年中国汽车半导体的市场规模为 39.99 亿美元，较 2012 年增长 22.2%，其中 MCU 的市场规模为 10.46 亿美元；到 2014 年，半导体的市场需求增长至 46.01 亿美元。

图 1-1 可以看到，未来几年，中国汽车电子及相关的半导体产业将保持稳步增长。其中，随着消费者对汽车安全性的关注日益提高，EPS、轮胎压力监测系统（Tire Pressure Monitoring System，TPMS），以及基于雷达和视觉的 ADAS 等一系列安全技术将逐步得到广泛应用，并成为今后一段时间的热点话题。而仪表显示与信息娱乐系统将在图形化仪表与大屏幕显示需求的推动下，继续成为汽车电子市场增长的引擎。随着国家各项节能减排政策的发布和落实，以混合动力（Hybrid Electric Vehicle，HEV）应用为主的新能源技术也将成为业界聚焦的热点。

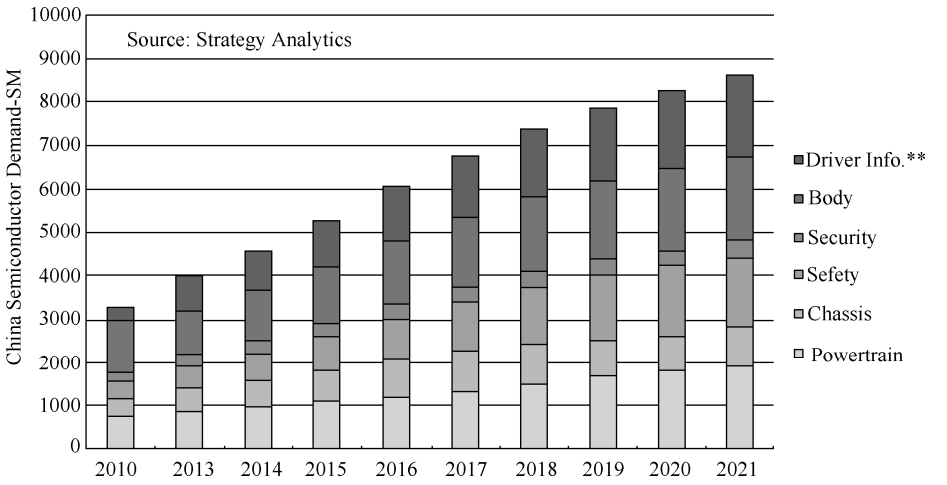


图 1-1 中国汽车半导体的市场规模（百万美元）

1.3 飞思卡尔在汽车电子市场中的地位

飞思卡尔是汽车电子以及嵌入式处理解决方案的领导者，其主要业务涉及汽车电子、消费电子、工业电子及网络设备等市场，其中，汽车电子类客户遍及全球各大顶尖零部件厂商和整车厂。

几十年来，凭借突破性思维、工程技术专长、对高品质的追求，以及长久保持的市场领军地位等优势，飞思卡尔在汽车电子市场上发挥着举足轻重的作用。

- 北美汽车半导体供应商领域：第一（来源 Strategy Analytics，2014 年 4 月）。
- 全球商用汽车 MEMS 传感器领域：第一（来源 HIS，2014 年 5 月）。
- 全球商用汽车加速度传感器领域：第一（来源 HIS，2014 年 5 月）。
- 全球汽车微控制器（Micro Controller Unit，MCU）领域：第二（来源 IHS，2014 年 3 月）。
- 全球车载信息娱乐处理器领域：第二（来源 Strategy Analytics，2014 年 4 月）。
- 飞思卡尔创造了全球第一款发动机控制芯片，并保持领先至今。
- 飞思卡尔是全球首家集成 77 GHz 雷达技术的汽车半导体供应商。
- 全球 TOP10 的汽车厂商有 7 家选择 i.MX6 应用处理器来支持其信息娱乐系统的连接和解决方案。
- 全球 6 大顶尖豪华车品牌将在其图形化仪表盘中采用 i.MX 应用处理器。

飞思卡尔拥有丰富的微控制器、处理器、模拟器件和传感器产品线，覆盖了从低端到高端的汽车电子产品，能够帮助客户实现全新的突破性汽车设计，包括动力总成、车身、底盘与安全性、车载信息娱乐和通信系统及车内网络应用。基于世界一流的架构，以及业界领先的安防和功能安全技术，飞思卡尔可以帮助客户实现开发的可靠性和自由性。凭借 Power Architecture 技术及 ARM 技术，客户可以获得多种设计选择。

飞思卡尔为汽车电子市场提供产品长期供货保障，广泛的器件都包含在该计划中，保证至少 15 年的供货期。飞思卡尔全面贯彻高品质企业文化，致力于保证产品质量、交货时间和最高服务水平，并成为客户眼中最卓越的半导体公司。

1.4　面向汽车电子的微控制器 KEA 系列 MCU 简介

Kinetis EA 系列（简称 KEA 系列）MCU 是飞思卡尔针对汽车市场最新开发的 MCU，有非常广泛的应用范围，具体的目标应用在本节有详细阐述。整个 KEA 系列引脚兼容，还将与未来的 Kinetis Auto 系列的其他系列产品引脚兼容，并提供多个引脚和存储选择，使开发人员能够在不同芯片之间进行移植，最大限度地提高硬件、软件重用性并减少投放市场的时间。

1．KEA 系列 MCU 的型号标识[①]

飞思卡尔 Kinetis 系列 MCU 的型号众多，但同一子系列的 CPU 核是相同的，多种型号只是为了适用于不同的应用场合。为了方便选型或订购，飞思卡尔给出 MCU 型号标识，Kinetis EA 系列型号标识格式为“QKEA A C FFF M T PP CCN”，其中，各字段说明如表 1-1 所示。本书使用的芯片型号为 SKEAZ128MLK，从该芯片型号标识可以获得如下信息：该芯片为汽车级、KEA 系列、M0+内核、包含 CAN 模块、程序 Flash 大小为 128 KB、运行温度范围是–40～125℃、80 引脚 LQFP 封装、CPU 最高频率为 48 MHz、盒包装。

① 来自《KEA 数据手册》第 2 章，该文档中给出的型号标识字段中含有程序存储器类型，实际出厂的芯片并未使用该字段，默认程序存储器为 Flash。

Kinetis EA 系列芯片命令字段说明如表 1-1 所示。

表 1-1 Kinetis EA 系列芯片命令字段说明

字 段	说 明	取 值
Q	质量状态	S=汽车级；P=工程测试芯片
KEA	Kinetis 汽车系列号	KEA
A	内核属性	Z=M0+内核
C	CAN 总线可用性	N=不包含 CAN 模块；（空）=包含 CAN 模块
FFF	程序 Flash 大小	64 = 64 KB；128=128 KB
M	生产版本	F0=第一版本；F1=第一版本之后的修订版（有时无字段）
T	运行温度范围	M= –40～125 ℃
PP	封装类型	LH =64LQFP(10 mm×10 mm)；LK =80LQFP(14 mm×14 mm)
CC	CPU 最高频率	4 = 48 MHz（有的芯片无该字段，默认 48 MHz）
N	包装类型	R=卷包装；（空）=盒包装

2．KEA 系列 MCU 的简明特性与结构框图

KEA 系列 MCU 基于的 ARM Cortex-M0+处理器是目前市场上能效最高的 32 位处理器，每微安数据吞吐量居业内领先水平。KEA 系列 MCU 具有多种灵活的超低功耗模式，适合不同的应用情形，可最大限度延长电池使用时间；在不唤醒内核的情况下，智能外设在深度睡眠模式下仍然可以工作，可进行智能决策并处理数据。KEA 系列 MCU 包含了一组功能强大的模拟、通信、定时和控制外设，提供各种闪存规格和引脚数。KEA 系列 MCU 简明特性如下，以下特性均可在图 1-2 中直观地看到。

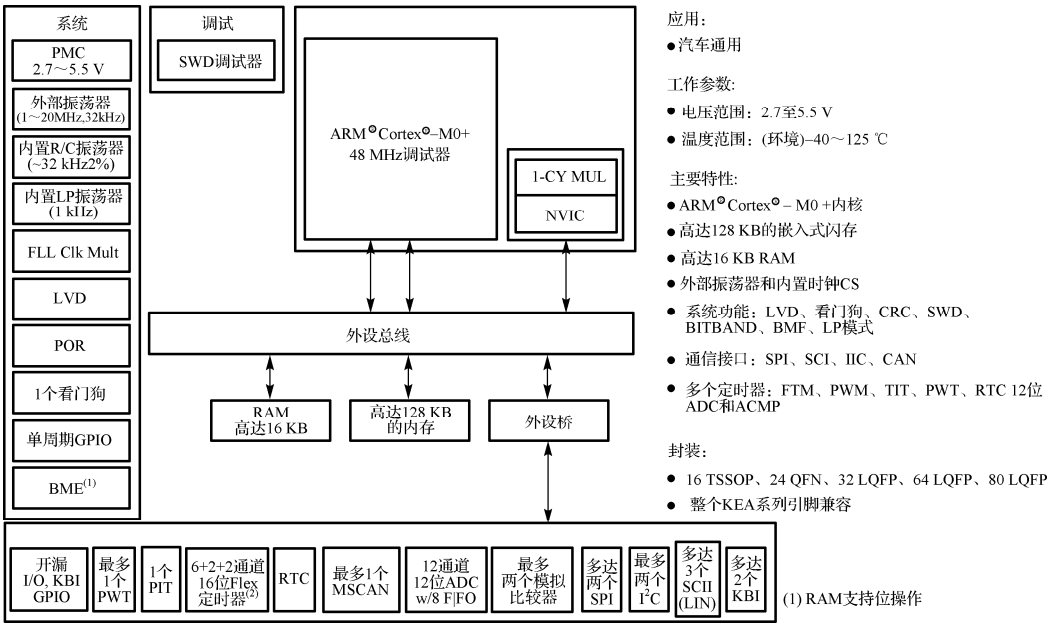


图 1-2 KEA 系列 MCU 结构框图

（1）MCU 内核运行频率高达 48 MHz，通过 AEC-Q100 一级认证，可满足温度范围为–40～125 ℃的严苛要求，并且增强了 ESD/EMC 性能。

（2）可扩展解决方案，包括高达 128 KB 的闪存及 80LQFP（71GPIO）的封装选项，并且可轻松扩展设计，以满足不同的汽车应用需求。

（3）工作电压范围为 2.7～5.5 V，可驱动高电流电机/组件，5 V 模拟/传感器组件可轻松连接至系统级设计。

（4）时钟模块：振荡器（Oscillator，OSC），支持 32.768 kHz 晶振或 4～24 MHz 晶体或陶瓷谐振器；内部时钟源（ICS），带有内部参考时钟（ICSIRCLK）和 FLL（Frequency Locked-Loop，锁频环），为 48 MHz 系统时钟提供 37.5 kHz 的预调整内部参考电压；内部带有 1 kHz 低功耗振荡器（Low Power Oscillator，LPO）。

（5）系统功能：电源管理模块（Power Management Module，PMM），具有 Run（运行）、Wait（等待）、Stop（停止）三种电源模式；低电压检测（Low-voltage detection，LVD），提供可选择的复位或中断触发；具有独立的时钟源看门狗（WDOG）；可编程循环冗余校验模块（CRC）；串行线调试接口（Serial Wire Debug，SWD）；位操作引擎（Bit Manipulation Engine，BME）。

（6）人机接口：具有 71 个通用输入/输出口（GPIO）；两个 32 位键盘中断模块（Keyboard Interrupt Modules，KBI）；外部中断（External Interrupt，IRQ）。

（7）模拟模块：一个 16 通道的 12 位 SAR ADC，可在停止模式下运行，提供硬件触发（ADC）选项；两个模拟比较器，包含一个 6 位 DAC 和可编程参考电压输入（ACMP）。

（8）定时器：一个 6 通道 FlexTimer/PWM (FTM)；两个 2 通道 FlexTimer/PWM (FTM)；一个 2 通道周期中断定时器（Periodic Interrupt Timer，PIT）；一个 16 位脉冲宽度定时器（Pulse Width Timer，PWT）；一个实时时钟（Real-Time Clock，RTC）。

（9）通信接口：两个 SPI 模块（串行外设接口）；三个 UART 模块；两个 I^2C 模块；一个 MSCAN 模块。可通过丰富的通信模块实现车内通信需求。

（10）快速原型工具可实现高质量软件开发，包括 Kinetis Design Studio、CodeWarrior IDE、Processor Expert 软件建模工具、MQX™ RTOS 和 AUTOSAR。

KEA 系列 MCU 的结构框图如图 1-2 所示。

3．KEA 系列 MCU 的简明资源列表

KEA 系列 MCU 由六个子系列组成，分别是内部不含 CAN 模块的 KEAZN8、KEAZN16、KEAZN32、KEAZN64，以及内部含 CAN 模块的 KEAZ64 和 KEAZ128，表 1-2 给出了 KEA 系列 MCU 的简明资源列表。所有 KEA 系列 MCU 均具有低功耗与丰富的混合信号控制外设，提供了不同的闪存容量和引脚数量，供实际应用选型。

4．KEA 系列 MCU 的相关参考设计

飞思卡尔提供了 KEA 系列 MCU 的相关参考资源，供读者实际应用时参考，这些资源可从飞思卡尔网站获得。

5．KEA 系列 MCU 的目标应用

KEA 系列 MCU 在汽车电子领域有着广泛的应用，例如，在信息娱乐系统连接模块、停

车辅助系统、DC/BLDC 电机控制、电子驻车制动、电池管理、泵/风扇控制器、智能无钥匙进入及启动系统、汽车照明、车身中控/空调、车窗/天窗/车门、座椅/后视镜/雨刮器、防盗装置、动力总成配套芯片、通用传感器节点等方面都可以作为目标应用。

表 1-2 KEA 系列 MCU 的简明资源列表

器件 / 资源	KEAZNx				KEAZx	
	KEAZN8	KEAZN16	KEAZN32	KEAZN64	KEAZ64	KEAZ128①
CPU 频率/MHz	48	40	40	40	48	48
RAM 容量/KB	1	2	4	4	8	16
Flash 容量/KB	8	16	32	64	64	128
GPIO 引脚数	22	57	57	57	71	71
UART	1	3	3	3	3	3
SPI	1	2	2	2	2	2
IIC	1	2	2	2	2	2
MSCAN	0	0	0	0	1	1
PWM	1	0	0	0	1	1
ACMP	2	2	2	2	2	2
12 位 ADC	12 通道	16 通道				
16 位 FTM	两个 FTM 模块，分别有 6 通道、2 通道	三个 FTM 模块，其中一个有 6 个通道，另两个均有 2 个通道				
封装	16TSSOP/24QFN	32/64LQFP			64/80LQFP	

表 1-3 KEA 系列 MCU 的相关参考设计

资源名称	特性	
TRK-KEA128	KEA128MCU 采用 80LQFP 封装；CAN 通信接口	板载 OpenSDA；LIN 通信接口；带有环境光传感器的模拟接口；4 个高效用户 LED；2 个按钮；SCI 串行通信接口（共享 OpenSDA 接口）
TRK-KEA64	KEA64MCU 采用 64LQFP 封装	
TRK-KEA8	KEA8MCU 采用 24LQFP 封装	
KEA128 三相无传感器 BLCD 电机控制参考设计	KEA128MCU 采用 80LQFP 封装，三相无传感器 BLCD 电机控制；通过电机控制应用调整（MCAT）工具进行实时算法参数调整；LIN 和 CAN 接口	
KEA128 LED 照明控制参考设计	KEA128MCU 采用 64LQFP 封装，照明控制解决方案；完整的头灯环境控制解决方案，包括头灯、LED 灯、尾灯、转向灯、灯电流控制和诊断；LIN 和 CAN 接口	
KEA128 汽车网参考设计	将 KEA128MCU 作为中央网关，显示其高效、精准的总线处理和节能功能	

① 本书选用。

第 2 章

ARM Cortex-M0+处理器

本章导读：KEA 系列 MCU 的内核使用 ARM Cortex-M0+处理器，需要学习 ARM Cortex-M0+汇编的读者可以阅读本章，一般读者简要了解 2.1 节即可。虽然本书使用 C 语言阐述 ARM Cortex-M0+ KEA 系列 MCU 的嵌入式开发，但理解 1 至 2 个结构完整、组织清晰的汇编程序对嵌入式开发将有很大帮助，也有助于更深层次的理解 ARM Cortex-M0+软件的设计。第 4 章中将结合 GPIO 的应用给出汇编实例，供学习参考。实际上，一些诸如初始化、操作系统调度、快速响应等特殊功能必须使用汇编完成。本章给出 ARM Cortex-M0+的特点、内核结构、存储器映像及内部寄存器概述；给出指令简表、寻址方式及指令的分类介绍；给出指令集与机器码对应表，供机器码级别的调试使用；给出 ARM Cortex-M0+汇编语言的基本语法。

本章参考资料：本章 2.1.1 节与 2.1.3 节参考自《M0+用户指南》；2.4 节参考自《GNU 汇编语法》、《Kinetis 汇编参考手册》及《M0+用户指南》。

2.1 ARM Cortex-M0+处理器简介

ARM Cortex-M0+系列处理器是 2012 年推出主要目标市场是 8 位/16 位微控制器的升级换代，具有性价比高、功耗低等特点。了解其特点、内核结构、存储器映像、内部寄存器、寻址方式及指令系统，可以为进一步学习和应用 ARM Cortex-M0+提供基础。

2012 年 3 月 14 日，ARM 公司于中国上海发布了一款拥有全球最低功耗的微处理器 ARM Cortex-M0+。该处理器采用低成本的 90 nm 低功耗（Low Power，LP）工艺，耗电量仅为 9 μA/MHz；该微处理器基于 ARMv6M 架构支持 Thumb 指令集[①]和部分 Thumb2 指令集[②]；加入多个重要新特性，包括单周期输入输出（IO）以加快通用输入输出（GPIO）和外围设备的存取速度、改良的调试和追踪能力、二阶流水线技术以减少每个指令所需的时钟周期数（CPI）和优化的闪存访问方式等，以进一步降低功耗。

Cortex-M0+处理器不仅延续了易用性、C 语言编程模型等优势，而且能够兼容已有的 Cortex-M0 处理器的工具。作为 Cortex-M 处理器系列中的一员，Cortex-M0+处理器同样可获得 ARM Cortex-M 整个系统的全面支持，其良好的软件兼容性，使其能够方便地被移植到更高性能的 Cortex-M3 或 Cortex-M4 等系列处理器。

① Thumb 是 ARM 体系结构中一种 16 位的指令集。

② Thumb2 是 ARM 体系结构中混合 32/16-bit 指令集。

2.1.1 ARM Cortex–M0+处理器特点与结构图

Cortex-M0+处理器组件结构图见图 2-1，图中虚线表示可选组件，下面简要介绍各部分。

1．Cortex-M0+内核

32 位 ARM Cortex-M0+处理器是以一个“处理器子系统”的形式出现的，其 CPU 内核本身与 NVIC 和一系列调试块都亲密耦合。Cortex-M0+处理器基于 2 级流水线冯·诺依曼架构[①]，内核版本为 ARMv6-M，支持 16 位 Thumb 指令集，同时采用 Thumb2 技术。Cortex-M0+内核的性能接近 8 位或 16 位竞争产品 CoreMark/mA[②]的 2 倍。

2．嵌套中断向量控制器（NVIC）

NVIC（Nested Vectored Interrupt Controller）是一个在 Cortex-M0+中内建的中断控制器，中断的具体路数由芯片厂商定义，本书使用的 MCU（飞思卡尔公司以 Cortex-M0+为内核设计的微处理器系列之一）共有 32 个可屏蔽外部中断源并支持 4 级优先级。NVIC 是与 CPU 紧耦合的，它还包含了若干个系统控制寄存器。因为 NVIC 支持中断嵌套，使得在 Cortex-M0+上处理嵌套中断时清晰而强大。它还采用了向量中断的机制，在中断发生时，会自动取出对应的服务例程入口地址，并且直接调用，无须软件判定中断源，缩短中断延时。为优化低功耗设计，NVIC 嵌套中断控制器还集成一个可选 WIC（唤醒中断控制器），在睡眠模式或深度睡眠模式下，芯片可快速进入超低功耗状态，且只能被 WIC 唤醒源唤醒。

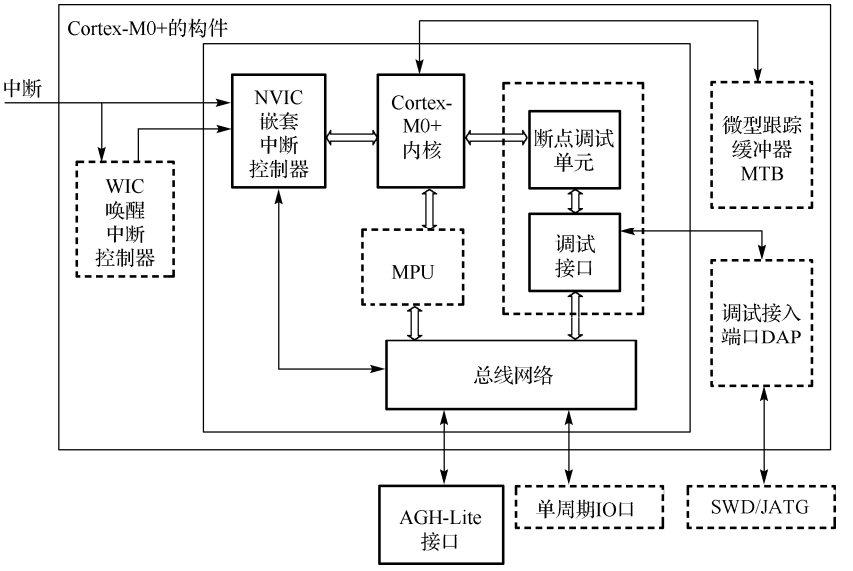


图 2-1 Cortex-M0+处理器结构图

① Cortex-M3/M4 采用哈佛结构，而 Cortex-M0+采用的是冯·诺依曼结构。区别在于它们是不是具有独立的程序指令存储空间和数据存储地址空间，如果有的话，就是哈佛结构，如果没有就是冯·诺依曼结构。而具有独立的地址空间也就意味着在地址总线和控制总线上至少要有一种总线必须是独立的，这样才能保证地址空间的独立性。

② CoreMark 是一项基准测试，它的目标就是要测试处理器核心性能。CoreMark 能分析并为处理器管线架构和效率评分，已成为量测与比较处理器性能的业界标准基准测试，CoreMark 数字越高，意味着更高的性能。

3．总线网络（BusMatrix）

BusMatrix 是 Cortex-M0+内部总线系统的核心。它是一个 AHB[①]互连的网络，通过它可以让数据在不同的总线之间并行传送（只要两个总线主机不试图访问同一块内存区域）。BusMatrix 还提供了附加的数据传送管理设施，包括一个写缓冲和一个按位操作的逻辑，这个按位操作的逻辑被称为位带（bit-band）。

4．调试组件

Cortex-M0+处理器实现了一个完全基于硬件的调试解决方案，该调试方案通过 2 针脚串行线协议（SWD）访问处理器、内存和外设，能够支持 2 个硬件断点和 2 个观察点；支持单步调试和向量捕捉；支持多个软件断点；通过总线网络非侵入式访问内核外设和零等待系统从机，调试器甚至能够在处理器运行时，访问包括内存在内的设备；在处理器停止状态时，可完全访问内核寄存器组。

5．总线接口

ARM Cortex-M0+处理器提供一个基于被称之为高级微控制器总线体系结构的总线规范 AMBA 技术的单一 32 位系统接口，可以高速整体访问所有系统外设和内存。所谓总线规范 AMBA（Advanced Microcontroller Bus Architecture），是一组针对基于 ARM 内核、片上系统之间通信而设计的标准协议。总线规范 AMBA 具有可选的 32 位单周期 I/O 接口，支持高速访问紧密耦合外设，如 GPIO 外设模块，该接口可从处理器、调试器以载入、存储方式访问，但不能执行代码；可选的 32 位从机接口，支持调试访问端口（Debug Access Port，DAP），该端口可通过串行或 JATG 协议调试；可选的内存保护单元接口；可选的执行跟踪接口（Execution Trace Interface，ETI），可配置执行跟踪缓冲区执行跟踪组件功能。在 AMBA 总线规范中，定义了 AHB、APB、ASB 三种总线。

6．SysTick 定时器

SysTick 定时器是一个 24 位倒计时定时器，即使系统在睡眠模式下也能工作，是作为嵌套中断向量控制器 NVIC 的一部分实现的，用于实时操作系统 RTOS 的时钟或仅仅作为计数器，一旦使能，定时器将从预设值减到 0，直到下个循环开始，标志位会被置位。

7．其他模块

系统控制块（System Control Block，SCB）提供了系统运行信息和系统配置功能，包括配置、控制、报告系统异常等。微型跟踪缓冲器（MTB）提供程序追踪功能，可以产生指令用来访问变化的数据。存储器保护单元（MPU）是一个选配的单元，本书中介绍的 KEA128 芯片不包含此组件。

2.1.2　ARM Cortex-M0+处理器存储器映像

ARM Cortex-M0+只有一个单一固定的存储器映射，这一点极大地方便了软件在各种 Cortex-M0+内核间的移植。举个简单的例子，各款 Cortex-M0+核 MCU 的 NVIC 和 MPU 都

① AHB（Advanced High Performace Bus）高性能总线，用于高性能系统模块的连接，支持突发模式数据传输和事务分割。

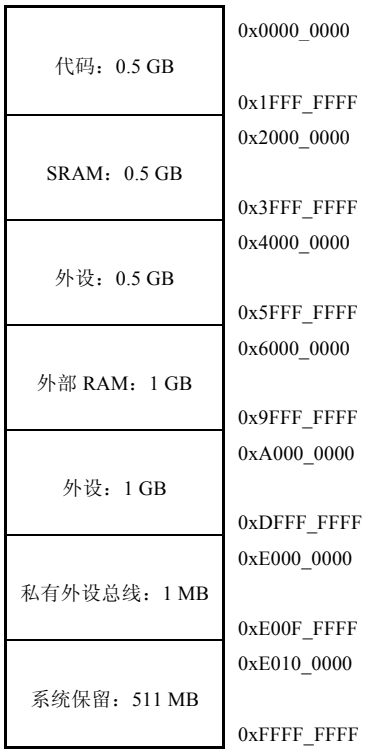


图 2-2 Cortex-M0+的存储器空间地址映像

在相同的位置布设寄存器，使得它们变得通用。尽管如此，Cortex-M0+定出的条条框框是粗线条的，它依然允许芯片制造商灵活地分配存储器空间，以制造出各具特色的 MCU 产品。图 2-2 给出了 M0+的存储器空间地址映像。

（1）它的存储器映射是预定义的，并且还规定好了哪个位置使用哪条总线。

（2）Cortex-M0+的存储器系统支持所谓的“位带”（bit-band）操作，通过它可以实现对单一比特的原子操作[①]。位带操作仅适用于一些特殊的存储器区域中。

（3）Cortex-M0+的存储器系统支持小端格式和大端格式[②]的配置，一般具体某款芯片在出厂时已经被厂商定义过。本书中 MCU 芯片被配置为小端格式[③]。

2.1.3 ARM Cortex-M0+处理器的寄存器

ARM Cortex-M0+处理器的寄存器有 R0～R15，如图 2-3 所示。其中 R13 作为堆栈指针 SP，SP 实质上有两个，但在同一时刻只能有一个可以看到，这也就是所谓的“banked”寄存器。特殊功能寄存器有预定义的功能，而且必须通过专用的指令来访问。

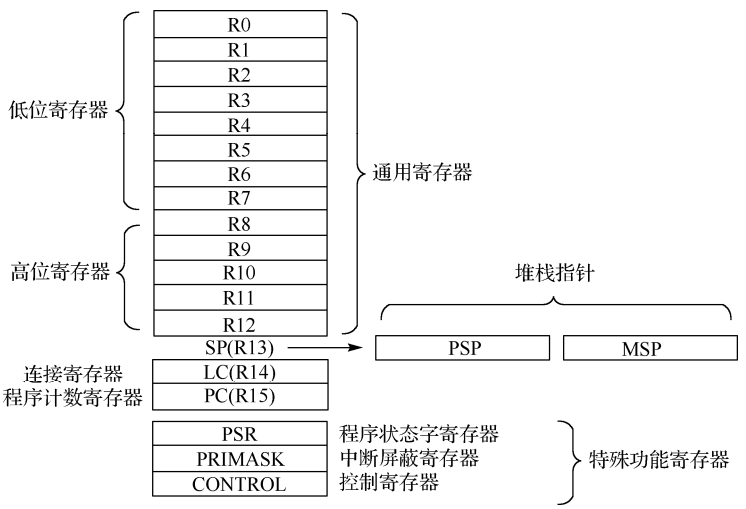


图 2-3 ARM Cortex-M0+处理器的寄存器组

① 详见《Cortex-M3 权威指南》，P83。

② 字节存储顺序（Endianess）分为小端格式（Little endian）和大端格式（Big endian），所谓小端格式是指字的低字节存储在低地址中，字的高字节存储在高地址中；大端格式则反之。

③ 参见《KEA 参考手册》3.3 节中表 3-5。

1．通用寄存器 R0～R12

R0～R12 是最具“通用目的”的 32 位通用寄存器，用于数据操作。32 位的 Thumb2 指令则可以访问所有通用寄存器。但绝大多数 16 位 Thumb 指令只能访问 R0～R7，因而 R0～R7 又称为低组寄存器，所有指令都能访问它们，它们的字长全是 32 位，复位后的初始值是随机的。R8～R12 也被称为高组寄存器，这是因为只有很少的 16 位 Thumb 指令能访问它们，32 位的指令则不受限制，它们也是 32 位字长，复位后的初始值是随机的。

2．堆栈指针 R13

R13 是堆栈指针，在 ARM Cortex-M0+处理器内核中共有两个堆栈指针，主堆栈指针 MSP 和进程堆栈指针 PSP，若用户用到其中一个，另一个必须用特殊指令来访问（MRS、MSR 指令），因此任一时刻只能使用其中的一个。主堆栈指针 MSP 是复位后缺省使用的堆栈指针，它可由操作系统内核、中断服务例程及所有需要特权访问的应用程序代码来使用；进程堆栈指针 PSP 用于常规的应用程序代码（不处于中断服务例程中时），该堆栈一般供用户的应用程序代码使用。要注意的是，并不是每个应用工程都用到这两个堆栈指针，简单的应用程序只用 MSP 就够了，并且 PUSH 指令和 POP 指令默认使用 MSP（有时 MSP 直接记为 SP）。另外，堆栈指针的最低两位永远是 0，即堆栈总是 4 字节对齐的。

3．连接寄存器 R14（LR）

当调用一个子程序时，由 R14 存储返回地址。不像大多数其他处理器那样，ARM 为了减少访问内存的次数（访问内存的操作往往要 3 个以上指令周期，带 MMU 和 cache 的就更加不确定了），把返回地址直接存储在寄存器中，这样足以使很多只有 1 级子程序调用的代码无须访问内存（堆栈内存），从而提高子程序调用的效率；如果多于 1 级，则需要把前一级的 R14 值压到堆栈里。在 ARM 上编程时，应尽量使用寄存器保存中间结果。

4．程序计数寄存器 R15（PC）

R15 是程序计数器，指向当前的程序地址。如果修改它的值，就能改变程序的执行流程（很多高级技巧隐藏其中）。在汇编代码中也可以使用名字“PC”来访问它。因为 ARM Cortex-M0+内部使用了指令流水线，读 PC 时返回的值是当前指令的地址+4。ARM Cortex-M0+中的指令至少是半字对齐的，所以 PC 的第 0 位总是 0。然而，在分支时，无论是直接写 PC 的值，还是使用分支指令，都必须保证加载到 PC 的数值是奇数（即第 0 位为 1），用以表明这是在 Thumb 状态下执行的，倘若第 0 位为 0，则视为企图转入 ARM 模式，ARM Cortex-M0+将产生异常。

5．特殊功能寄存器

Cortex-M0+内核包括了一组特殊功能寄存器，包括程序状态字寄存器组（xPSR）、中断屏蔽寄存器（PRIMASK）和控制寄存器（CONTROL）。

1）程序状态字寄存器（xPSR）

程序状态字寄存器在内部分为以下几个子寄存器：APSR、IPSR、EPSR，用户可以使用

MRS 和 MSR 指令访问寄存器。三个子寄存器既可以单独访问，也可以两个或三个组合到一起访问。使用三合一方式访问时，把该寄存器称为 xPSR，见表 2-1。

表 2-1 ARM Cortex-M0+程序状态寄存器（xPSR）

数据位	31	30	29	28	27～25	24	23～10	9	8～6	5	4	3	2	1	0
APSR	N	Z	C	V											
IPSR										异常号					
EPSR						T									
xPSR	N	Z	C	V		T				异常号					

（1）APSR 寄存器：显示算术运算单元 ALU 状态位的一些信息。

负标志 N：若结果最高位为 1，相当于有符号运算中结果为负，则置 1，否则清 0。

零标志 Z：若结果为 0，则置 1，否则清 0。

进位标志 C：若有最高位的进位（减法为借位），则置 1，否则清 0。

溢出标志 V：若溢出，则置 1，否则清 0。

这些位在条件转移指令中被用到，复位之后这些位是随机的。

（2）IPSR 寄存器：每次异常完成之后，处理器会实时更新 IPSR 内的异常号。只能被 MRS 指令读写。进程模式下，值为 0；Handler 模式[①]下，存放当前异常的异常号。复位之后，寄存器被自动清 0。复位异常号是一个暂时值，复位时，是不可见的。

（3）EPSR 寄存器：T 标志位指示当前运行的是否 Thumb 指令，该位是不能被软件读取的。运行复位向量对应的代码时置 1。如果该位为 0，会发生硬件异常，进入硬件中断服务例程。

2）中断屏蔽寄存器（PRIMASK）

中断屏蔽寄存器的 D31～D1 位保留，只有 D0 位（记为 PM）有意义。当该位被置位时，除不可屏蔽中断和硬件错误之外的所有中断都被屏蔽。使用特殊指令（如 MSR、MRS）可以访问该寄存器，还有一条称为改变处理器状态的特殊指令 CPS 也能访问它。只有在实时任务时才会用到。执行汇编指令“CPSID i”，将 D0 位置 1（关总中断）；执行汇编指令“CPSIE i”，将 D0 位清 0（开总中断），其中 i 代表 IRQ 中断，IRQ 是非内核中断请求（Interrupt Request）的缩写。

3）控制寄存器（CONTROL）

内核中的控制寄存器（CONTROL）的 D31～D2 位保留，D1、D0 位含义如下。

D1（SPSEL）：堆栈指针选择位。SPSEL=0，使用主堆栈指针 MSP 为当前堆栈指针（复位后缺省值）；SPSEL=1，在线程模式下，使用线程堆栈 PSP 指针为当前堆栈指针。在特权、线程模式下，软件可以更新 SPSEL 位；在 Handler 模式下，写该位无效。复位后，控制寄存器清 0。可用 MRS 指令读该寄存器，MSR 指令写该寄存器。非特权访问无效。

D0（nPRIV）：如果权限扩展，在线程模式下定义执行特权。nPRIV= 0，线程模式下可以特权访问；nPRIV=1，线程模式下无特权访问。在 Handler 模式下，总是特权访问。

① Handler 模式：处理异常的模式。

2.2　ARM Cortex-M0+处理器的指令系统

CPU 的功能是从外部设备获得数据，通过加工、处理，再把处理结果送到 CPU 的外部世界。设计一个 CPU，首先需要设计一套可以执行特定功能的操作命令，这种操作命令称为指令。CPU 所能执行的各种指令的集合，称为该 CPU 的指令系统。表 2-2 给出了 ARM Cortex-M 指令集概况。

表 2-2　ARMCortex-M 指令集概况

ARM Cortex-M	Thumb	Thumb-2	硬 件 乘 法	硬件除法	饱和运算	DSP 扩展	浮点	ARM 架构	核心架构
Cortex-M0	大部分	子集	1 或 32 个周期	无	无	无	无	ARMv6-M	冯・诺依曼
Cortex-M0+	大部分	子集	1 或 32 个周期	无	无	无	无	ARMv6-M	冯・诺依曼
Cortex-M1	大部分	子集	3 或 33 个周期	无	无	无	无	ARMv6-M	冯・诺依曼
Cortex-M3	全部	全部	1 个周期	有	有	无	无	ARMv7-M	哈佛
Cortex-M4	全部	全部	1 个周期	有	有	有	可选	ARMv7E-M	哈佛

ARM Cortex-M0+处理器将上一代 Cortex-M0 升级为真正的 8 位替代产品，同时保留了与所有其他 Cortex-M 级处理器之间的兼容性。其指令系统是 ARM Cortex-M3/M4 指令集的一部分，并且与 ARM Cortex-M0 完全兼容，这允许设计人员可重复利用其现有的编译器和调试工具。

2.2.1　ARM Cortex–M0+指令简表与寻址方式

1．ARM Cortex-M0+指令简表

学习和记忆 ARM Cortex-M0+基本指令对理解处理器特性十分有益的，这里给出的 ARM Cortex-M0+指令简表可以方便读者记忆基本指令。

ARM Cortex-M0+共有 57 条基本指令，依据不同的寻址方式形成 68 条具体指令。为了方便学习，将这些指令分为数据传送类，数据操作类（算术运算、逻辑运算、位操作、反转字节、扩展字节和移位），跳转类和其他指令等四大类指令。本节分别介绍这些指令，并对每条具体指令进行统一编号。表 2-3 给出保留字的简明含义，供读者了解和记忆 ARM Cortex-M0+指令。

2．ARM Cortex-M0+的寻址方式

指令是对数据的操作，通常把指令中所要操作的数据称为操作数，ARM Cortex-M0+处理器所需的操作数可能来自寄存器、指令代码、存储单元。而确定指令中所需操作数的各种方法称为寻址方式（Addressing Mode）。下面的指令格式中的“{}”表示其中可选项，如“LDRH Rt, [Rn {, #imm}]”，表示有“LDRH　Rt, [Rn]”、“LDRH　Rt, [Rn , #imm]”两种指令格式。指令中的“[]”表示其中内容为地址，“@”表示注释。

（1）立即数寻址。在立即数寻址方式中，操作数直接通过指令给出，数据包含在指令编码中，随着指令一起被编译成机器码存储于程序空间中。用“#”作为立即数的前导标识符，ARM Cortex-M0+的立即数范围是 0x00～0xff。例如：

表 2-3 ARM Cortex-M0+指令简表

类 型		保 留 字	含 义
数据传送类		ADR	生成与 PC 指针相关的地址
		LDR、LDRH、LDRB、LDRSB、LDRSH、LDM	存储器中内容加载到寄存器中
		STR、STRH、STRB、STM	寄存器中内容存储至存储器中
		MOV、MVN	寄存器间数据传送指令
		POP、PUSH	进栈、出栈
数据操作类	算术运算类	ADC、ADD、RSB、SBC、SUB、MUL	加、减、乘指令
		CMN、CMP	比较指令
	逻辑运算类	AND、ORR、EOR、BIC	按位与、或、异或、位段清零
	数据序转类	REV、REV16、REVSH	反转字节序
	扩展类	SXTB、SXTH、UXTB、UXTH	无符号扩展字节、有符号扩展字节
	位操作类	TST	测试位指令
	移位类	ASR、LSL、LSR、ROR	算术右移、逻辑左移、逻辑右移、循环右移
跳转控制类		B{cc}、BL、BX、BLX	跳转指令
其他指令		BKPT、CPSID、CPSIE、DSB、DMB、ISB、MRS、MSR、NOP、SEV、SVC、WFE、WFI	

```
SUBR1,R0, #1          @R1←R0-1
MOVR0,   #0xff        @将立即数 0xff 装入 R0 寄存器
```

（2）寄存器寻址。在寄存器寻址中，操作数来自于寄存器。例如：

```
MOVR1,R2              @R1←R2
SUB  R0,R1,R2         @R0←R1-R2
```

（3）直接寻址。在直接寻址方式中，操作数来自于存储单元，指令中直接给出存储单元地址。在指令码中，显示给出数据的位数，有字（4 字节）、半字（2 字节）、单字节三种情况。例如：

```
LDR   Rt,label        @从标号 label 处连续取 4 字节至 Rt 寄存器中
LDRH  Rt,label        @从地址 label 处读取半字到 Rt
LDRB  Rt,label        @从地址 label 处读取字节到 Rt
```

（4）偏移寻址及寄存器间接寻址。在偏移寻址中，操作数来自于存储单元，指令中通过寄存器及偏移量给出存储单元的地址。偏移量不超过 4 KB（指令编码中偏移量为 12 位），偏移量为 0 的偏移寻址也称为寄存器间接寻址。例如：

```
LDR R3, [PC, #2]      @地址为（PC+2）的存储器单元内容加载到 R3 中
LDR R3,[R4]           @地址为 R4 的存储单元内容加载到 R3 中
```

2.2.2 数据传送类指令

1. 生成与指针 PC 相关地址指令

ADR 指令（见表 2-4）将指针 PC 值加上一个偏移量得到的地址写进目标寄存器中。由于该地址只与指针 PC 相关，即 ADR 生成的是与位置无关代码。若利用 ADR 指令生成的目

标地址用于跳转指令 BX、BLX，则必须确保该地址最后一位为 1。Rd 为目标寄存器，label 为与指针 PC 相关的表达式。在该指令下，Rd 必须为 R0～R7，数值必须字对齐且在当前 PC 值的 1020 字节以内。此指令不影响 N、Z、C、V 状态标志。

表 2-4　ADR 指令

编　　号	指　　令	说　　明
(1)	ADR　Rd, label	生成与指针 PC 相关地址，将 label 的相对于当前指令的偏移地址值与 PC 相加或者相减后写入 Rd 中

2．取数指令

存储器中内容加载到寄存器中的指令见表 2-5，其中 LDR、LDRH、LDRB 指令分别表示加载来自存储器单元的一个字、半字和单字节（不足部分以 0 填充）。LDRSH 和 LDRSB 指令指加载存储单元的半字、字节有符号数扩展成 32 位到指定寄存器 Rt。

表 2-5　取数指令

编号	指　　令	说　　明
(2)	LDR　Rt, [<Rn \| SP> {, #imm}]	从{SP/Rn+ #imm}地址处，取字到 Rt，imm=0,4,8,…,1020
	LDR　Rt,[Rn, Rm]	从地址 Rn+Rm 处读取字到 Rt
	LDR　Rt, label	从 label 指定的存储器单元取数至寄存器
(3)	LDRH　Rt, [Rn {, #imm}]	从{Rn+ #imm}地址处，取半字到 Rt 中，imm=0,2,4,…,62
	LDRH　Rt,[Rn, Rm]	从地址 Rn+Rm 处读取半字到 Rt
(4)	LDRB　Rt, [Rn {, #imm}]	从{Rn+ #imm}地址处，取字节到 Rt 中，imm=0～31
	LDRB　Rt,[Rn, Rm]	从地址 Rn+Rm 处读取字节到 Rt
(5)	LDRSH　Rt,[Rn, Rm]	从地址 Rn+ Rm 处读取半字至 Rt，并带符号扩展至 32 位
(6)	LDRSB　Rt,[Rn, Rm]	从地址 Rn+ Rm 处读取字节至 Rt，并带符号扩展至 32 位
(7)	LDM　Rn{!}, reglist	从 Rn 处读取多个字，加载到 reglist 列表寄存器中，每读一个字后 Rn 自增一次

在“LDM Rn{!},reglist”指令中，Rn 表示存储器单元起始地址的寄存器；reglist 若包含一个或多个寄存器，若包含多个寄存器必须以“,”分隔，外面用“{}”标识；“!”是一个可选的回写后缀， reglist 列表中包含了 Rn 寄存器时不要回写后缀，否则须带回写后缀“!”。带后缀时，在数据传送完毕之后，将最后的地址将写回到 Rn=Rn+4×(n–1)，n 为 reglist 中寄存器的个数。Rn 不能为 R15，reglist 可以为 R0～R15 任意组合；Rn 寄存器中的值必须字对齐。这些指令不影响 N、Z、C、V 状态标志。

3．存数指令

寄存器中内容存储至存储器中的指令见表 2-6。STR、STRH 和 STRB 指令存储 Rt 寄存器中的字、低半字或低字节至存储器单元，存储器单元地址由 Rn 与 Rm 之和决定，Rt、Rn 和 Rm 必须为 R0～R7 之一。

其中，“STM Rn!, reglist 指令将 reglist 列表寄存器内容以字存储至 Rn 寄存器中的存储单元地址。以 4 字节访问存储器地址单元，访问地址从 Rn 寄存器指定的地址值到 Rn+4×(n–1)，n 为 reglist 中寄存器的个数。按寄存器编号递增顺序访问，最低编号使用最低地址空间，最高编号使用最高地址空间。对于 STM 指令，若 reglist 列表中包含了 Rn 寄存器，则 Rn 寄存器必须位于列表首位。如果列表中不包含 Rn，则将位于 Rn+4×n 地址回写到 Rn 寄存器中。这些指令不影响 N、Z、C、V 状态标志。

表 2-6 存数指令

编 号	指 令	说 明
(8)	STR Rt, [<Rn \| SP> {, #imm}]	把 Rt 中的字存储到地址 SP/Rn+#imm，imm=0,4,8,…,1020
	STR Rt, [Rn, Rm]	把 Rt 中的字存储到地址 Rn+ Rm 处
(9)	STRH Rt, [Rn {, #imm}]	把 Rt 中的低半字存储到地址 SP/Rn+#imm，imm=0,2,4,…,62
	STRH Rt, [Rn, Rm]	把 Rt 中的低半字存储到地址 Rn+ Rm 处
(10)	STRB Rt, [Rn {, #imm}]	把 Rt 中的低字节 SP/Rn+#imm。imm=0～31
	STRB Rt, [Rn, Rm]	把 Rt 中的低字节存储到地址 Rn+ Rm 处
(11)	STM Rn!, reglist	存储多个字到 Rn 处。每存一个字后 Rn 自增一次

4．寄存器间数据传送指令

MOV 指令（见表 2-7），Rd 表示目标寄存器；imm 为立即数，范围为 0x00～0xff。当 MOV 指令中 Rd 为 PC 寄存器时，丢弃第 0 位；当出现跳转，传送值的第 0 位清 0 后的值作为跳转地址。虽然 MOV 指令可以用作分支跳转指令，但强烈推荐使用 BX 或 BLX 指令。这些指令影响 N、Z 状态标志，但不影响 C、V 状态标志。

表 2-7 寄存器间数据传送指令

编 号	指 令	说 明
(12)	MOV Rd, Rm	Rd←Rm, Rd 只可以是 R0～R7。 MOVS 指令功能与 MOV 相同，且影响 N、Z 标志将寄存器 Rm 中数据取反，传送给寄存器 Rd，影响 N、Z 标志
(13)	MOVS Rd, #imm	
(14)	MVN Rd, Rm	

5．堆栈操作指令

堆栈操作指令见表 2-8。PUSH 指令将寄存器值存于堆栈中，最低编号寄存器使用最低存储地址空间，最高编号寄存器使用最高存储地址空间；POP 指令将值从堆栈中弹回寄存器，最低编号寄存器使用最低存储地址空间，最高编号寄存器使用最高存储地址空间。执行 PUSH 指令后，更新 SP 寄存器值 SP=SP–4；执行 POP 指令后更新 SP 寄存器值 SP=SP+4。若 POP 指令的 reglist 列表中包含了 PC 寄存器，在 POP 指令执行完成时跳转到该指针 PC 所指地址处。该值最低位通常用于更新 xPSR 的 T 位，此位必须置 1 确保程序正常运行。

表 2-8 堆栈操作指令

编 号	指 令	说 明
(15)	PUSH reglist	进栈指令。SP 递减 4
(16)	POP reglist	出栈指令。SP 递增 4

例如：

```
PUSH {R0,R4-R7}      @将 R0,R4～R7 寄存器值入栈
PUSH {R2,LR}         @将 R2,LR 寄存器值入栈
POP {R0,R6,PC}       @出栈值到 R0,R6,PC 中，同时跳转至 PC 所指向的地址
```

2.2.3 数据操作类指令

1．算术运算类指令

算术类指令有加、减、乘、比较等，见表 2-9。

表 2-9　算术类指令

编号	指　令	说　明
(17)	ADC　{Rd, } Rn, Rm	带进位加法。Rd←Rn+Rm+C，影响 N、Z、C 和 V 标志位
(18)	ADD　{Rd } Rn, <Rm \| #imm>	加法。Rd←Rn+Rm，影响 N、Z、C 和 V 标志位
(19)	RSB　{Rd, } Rn, #0	Rd←0-Rn，影响 N、Z、C 和 V 标志位（CW 环境不支持）
(20)	SBC　{Rd, }Rn, Rm	带借位减法。Rd←Rn-Rm-C，影响 N、Z、C 和 V 标志位
(21)	SUB　{Rd } Rn, <Rm \| #imm>	常规减法。Rd←Rn-Rm/ #imm，影响 N、Z、C 和 V 标志位
(22)	MUL　Rd, Rn Rm	常规乘法，Rd←Rn*Rm，同时更新 N、Z 状态标志，不影响 C、V 状态标志。该指令所得结果与操作数是否为无符号、有符号数无关。Rd、Rn、Rm 寄存器必须为 R0～R7，且 Rd 与 Rm 须一致。
(23)	CMN　Rn, Rm	加比较指令。Rn+Rm，更新 N、Z、C 和 V 标志，但不保存所得结果。Rn、Rm 寄存器必须为 R0～R7
(24)	CMP　Rn, #imm CMP　Rn, Rm	（减）比较指令。Rn-Rm/#imm，更新 N、Z、C 和 V 标志，但不保存所得结果。Rn、Rm 寄存器为 R0～R7，立即数 imm 范围 0～255

加、减指令对操作数的限制条件见表 2-10。

表 2-10　ADC、ADD、RSB、SBC 和 SUB 操作数限制条件

指　令	Rd	Rn	Rm	imm	限 制 条 件
ADC	R0～R7	R0～R7	R0～R7	—	Rd 和 Rn 必须相同
ADD	R0～R15	R0～R15	R0～PC	—	Rd 和 Rn 必须相同；Rn 和 Rm 不能同时指定为 PC 寄存器
	R0～R7	SP或PC	—	0～1020	立即数必须为4的整数倍
	SP	SP	—	0～508	立即数必须为4的整数倍
ADD	R0～R7	R0～R7	—	0～7	—
	R0～R7	R0～R7	—	0～255	Rd和Rn必须相同
	R0～R7	R0～R7	R0～R7	—	—
RSB	R0～R7	R0～R7	—	—	—
SBC	R0～R7	R0～R7	R0～R7	—	Rd 和 Rn 必须相同
SUB	SP	SP	—	0～508	立即数必须为 4 的整数倍
SUB	R0～R7	R0～R7	—	0～7	—
	R0～R7	R0～R7	—	0～255	Rd和Rn必须相同
	R0～R7	R0～R7	R0～R7	—	—

2．逻辑运算类指令

逻辑运算类指令见表 2-11。AND、EOR 和 ORR 指令把寄存器 Rn、Rm 值逐位与、异或和或操作；BIC 指令将寄存器 Rn 的值与 Rm 的值的反码按位作逻辑“与”操作，结果保存到 Rd。这些指令更新 N、Z 状态标志，不影响 C、Z 状态标志。

Rd、Rn 和 Rm 必须为 R0～R7，其中 Rd 为目标寄存器，Rn 为存放第一个操作数寄存器且必须和目标寄存器 Rd 一致（即 Rd 就是 Rn），Rm 为存放第二个操作数寄存器。

表 2-11　逻辑运算类指令

编　号	指　令	说　明	举　例
(25)	AND　{Rd, } Rn, Rm	按位与	AND R2, R2, R1
(26)	ORR　{Rd, } Rn, Rm	按位或	ORR R2, R2, R5
(27)	EOR　{Rd, } Rn, Rm	按位异或	EOR R7, R7, R6
(28)	BIC　{Rd, } Rn, Rm	位段清零	BIC R0, R0, R1

3．数据序转指令

数据序转指令见表 2-12。

表 2-12 数据序转指令

编　号	指　令	说　明
(29)	REV　Rd, Rn	将32位大端数据转小端存放或将32位小端数据转大端存放
(30)	REV16　Rd, Rn	将一个32位数据划分成两个16位大端数据，将这两个16位大端数据转小端存放或将一个32位数据划分成两个16位小端数据，将这两个16位小端数据转大端存放
(31)	REVSH　Rd, Rn	将16位带符号大端数据转成32位带符号小端数据或将16位带符号小端数据转成32位带符号大端数据

该指令用于改变数据的字节顺序。Rn 为源寄存器，Rd 为目标寄存器，且必须为 R0～R7 之一。REV 指令将 32 位大端数据转小端存放或将 32 位小端数据转大端存放；REV16 指令将一个 32 位数据划分成两个 16 位大端数据，将这两个 16 位大端数据转小端存放或将一个 32 位数据划分成两个 16 位小端数据，将这两个 16 位小端数据转大端存放；REVSH 指令将 16 位带符号大端数据转成 32 位带符号小端数据或将 16 位带符号小端数据转成 32 位带符号大端数据，如图 2-4 所示。这些指令不影响 N、Z、C、V 状态标志。

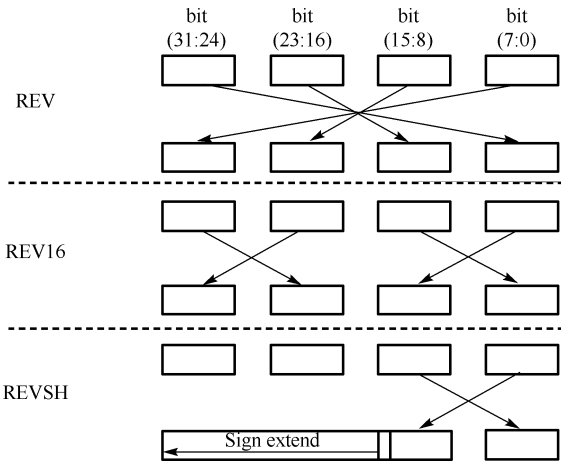


图 2-4 反序操作

4．扩展类指令

扩展类指令见表 2-13。寄存器 Rm 存放待扩展操作数；寄存器 Rd 为目标寄存器；Rm、Rd 必须为 R0～R7。这些指令不影响 N、Z、C、V 状态标志。

表 2-13 扩展类指令

编　号	指　令	说　明
(32)	SXTB　Rd, Rm	将操作数Rm的Bit[7:0]带符号扩展到32位，结果保存到Rd中
(33)	SXTH　Rd, Rm	将操作数Rm的Bit[15:0]带符号扩展到32位，结果保存到Rd中
(34)	UXTB　Rd, Rm	将操作数Rm的Bit[7:0]无符号扩展到32位，结果保存到Rd中
(35)	UXTH　Rd, Rm	将操作数Rm的Bit[15:0]无符号扩展到32位，结果保存到Rd中

5．位测试指令

位操作类指令见表 2-14。

表 2-14　位测试指令

编　号	指　令	说　明
(36)	TST　Rn, Rm	将Rn寄存器值逐位与Rm寄存器值进行与操作，但不保存所得结果。为测试寄存器Rn某位为0或1，将Rn寄存器某位置1，其余位清0。寄存器Rn、Rm必须为R0～R7。该指令根据结果，更新N、Z状态标志，但不影响C、V状态标志

6．移位类指令

移位类指令见表 2-15。ASR、LSL、LSR 和 ROR 指令，将寄存器 Rm 值由寄存器 Rs 或立即数 imm 决定移动位数，执行算术右移、逻辑左移、逻辑右移和循环右移。这些指令中，Rd、Rm、Rs 必须为 R0～R7。对于非立即数指令，Rd 和 Rm 必须一致。Rd 为目标寄存器，若省去 Rd，表示其值与 Rm 寄存器一致；Rm 为存放被移位数据寄存器；Rs 为存放移位长度寄存器；imm 为移位长度，ASR 指令移位长度范围 1～32，LSL 指令移位长度范围 0～31，LSR 指令移位长度范围 1～32。

表 2-15　移位指令

编　号	指　令	操　作	举　例
(37)	ASR　{Rd, } Rm, Rs ASR　{Rd, } Rm, #imm	┌─┐ ————→ └→□□□□……□□□□→C b31 b0	算术右移 ASR R7, R5, #9
(38)	LSL　{Rd, } Rm, Rs LSL　{Rd, } Rm, #imm	←———————— C←□□□□……□□□□←0 b31b0	逻辑左移 LSL R1, R2, #3
(39)	LSR　{Rd, } Rm, Rs LSR　{Rd, } Rm, #imm	————————→ 0→□□□□……□□□□→C b31 b0	逻辑右移 LSRR1, R2, #3
(40)	ROR　{Rd, } Rm, Rs	┌———————————————┐ └——→□□□□……□□□□→C-┘ b31b0	循环右移 ROR R4, R4, R6

（1）单向移位指令。算术右移指令 ASR 指令比较特别，它把要操作的字节当做有符号数，而符号位（b31）保持不变，其他位右移一位，即首先将 b0 位移入 C 中，其他位（b1～b31）右移一位，相当于操作数除以 2。为了保证符号不变，ASR 指令使符号位 b31 返回本身。逻辑右移指令 LSR 把 32 位操作数右移一位，首先将 b0 位移入 C 中，其他右移一位，0 移入 b31。根据结果，ASR、LSL、LSR 指令对标志位 N、Z 有影响；最后移出位更新 C 标志位。

（2）循环移位指令。在循环右移指令 ROR 中，将 b0 位移入 b31 中的同时也移入 C 中，其他位右移一位，从 b31～b0 内部看来循环右移了一位。根据结果，ROR 指令对标志位 N、Z 有影响；最后移出位更新 C 标志位。

2.2.4　跳转控制类指令

跳转控制类指令见表 2-16 ，这些指令不影响 N、Z、C、V 状态标志。

表 2-16　跳转控制类指令

编　号	指　令	跳 转 范 围	说　明
(41)	B{cond}　label	–256～+254 B	转移到 label 处对应的地址处。可以带（或不带）条件，所带条件见表 2-17，如 BEQ 表示标志位 Z=1 时转移

续表

编　号	指　　令	跳 转 范 围	说　　明
(42)	BL　label	–16～+16 MB	转移到 label 处对应的地址，并且把转移前的下条指令地址保存到 LR，并置寄存器 LR 的 Bit[0]为 1，保证了随后执行 POP {PC}或 BX 指令时成功返回分支
(43)	BX　Rm	任意	转移到由寄存器 Rm 给出的地址，寄存器 Rm 的 Bit[0]必须为 1，否则会导致硬件故障
(44)	BLX　Rm	任意	转移到由寄存器 Rm 给出的地址，并且把转移前的下条指令地址保存到 LR。寄存器 Rm 的 Bit[0]必须为 1，否则会导致硬件故障

```
BEQ label        @条件转移，标志位 Z=1 时转移到 label
BL funC          @呼叫子程序 funC，把转移前的下条指令地址保存到 LR
BX LR            @返回到函数调用处
```

表 2-17　B 指令所带的条件

条件后缀	标　志　位	含　　义	条件后缀	标　志　位	含　　义
EQ	Z=1	相等	HI	C=1 并且 Z=0	无符号数大于
NE	Z=0	不相等	LS	C=1 或 Z=1	无符号数小于或等于
CS 或者 HS	C=1	无符号数大于或等于	GE	N=V	带符号数大于或等于
CC 或者 LO	C=0	无符号数小于	LT	N!=V	带符号数小于
MI	N=1	负数	GT	Z=0 并且 N=V	带符号数大于
PL	N=0	正数或零	LE	Z=1 并且 N!=V	带符号数小于或等于
VS	V=1	溢出	AL	任何情况	无条件执行
VC	V=0	未溢出			

2.2.5　其他指令

其他指令见表 2-18 所示，，spec_reg 表示特殊寄存器 APSR、IPSR、EPSR、IEPSR、IAPSR、EAPSR、PSR、MSP、PSP、PRIMASK 或者 CONTROL。休眠指令 WFE 执行情况由事件寄存器决定。若事件寄存器为零，只有在发生如下事件才执行。

- 发生异常，且该异常未被异常屏蔽寄存器或当前优先级屏蔽；
- 在进入异常期间，系统控制寄存器的 SEVONPEND 置位；
- 若使能调试模式时，触发调试请求；
- 外围设备发出一个事件或在多重处理器系统中另一个处理器使用 SVC 指令。

若事件寄存器为 1，WFE 指令清该寄存器后立刻执行。在程序运行中，WFE 只用于低功耗模式，并不产生其他操作，类似于 NOP 指令。

休眠指令 WFI 执行条件为：发生异常，或 PRIMASK.PM 被清 0，产生的中断将会先占，或发生触发调试请求（不论调试是否被使能）。WFI 指令只在低功耗运行，在程序运行中，WFI 只用于低功耗模式，并不产生其他操作，类似于 NOP 指令。

表 2-18　其他指令

类　　型	编号	指　　令	说　　明
断点指令	(45)	BKPT　#imm	如果调试被使能，则进入调试状态（停机），者如果调试监视器异常被使能，则调用一个调试异常，否则调用一个错误异常。处理器忽视立即数 imm，立即数范围为～255，表示断点调试的信息。该指令不影响 N、Z、C、V 状态标志

续表

类　型	编号	指　令	说　明
中断指令	(46)	CPSIE　i	除了 NMI（不可屏蔽中断），禁止总中断，不影响 N、Z、C、V 状态标志
	(47)	CPSID　i	除了 NMI（不可屏蔽中断），使能总中断，不影响 N、Z、C、V 状态标志
屏蔽指令	(48)	DMB	数据内存屏蔽（与流水线、MPU 和 cache 等有关）
	(49)	DSB	数据同步屏蔽（与流水线、MPU 和 cache 等有关）
	(50)	ISB	指令同步屏蔽（与流水线、MPU 等有关）
特殊寄存器操作指令	(51)	MRS　Rd, spec_reg1	加载特殊功能寄存器的值到通用寄存器。若当前执行模式不为特权模式，除了 APSR 寄存器外，读其余所有寄存器值均为 0
	(52)	MSR　spe_reg, Rn	存储通用寄存器的值到特殊功能寄存器。Rd 不允许为 SP 或 PC 寄存器，若当前执行模式不为特权模式，除 APSR 寄存器外，任何试图修改寄存器操作均被忽视。影响 N、Z、C、V 标志
空操作	(53)	NOP	空操作，但无法保证能够延迟时间，处理器可能在执行阶段之前就将此指令从线程中移除。不影响 N、Z、C、V 标志
发送事件指令	(54)	SEV	发送事件指令。SEV 指令在多处理器系统中，向所有处理器发送一个事件，也可置位本地寄存器。不影响 N、Z、C、V 状态标志
操作系统服务调用指令	(55)	SVC　#imm	操作系统服务调用，带立即数调用代码。SVC 指令触发 SVC 异常。处理器忽视立即数 imm，若需要，该值可通过异常处理程序重新取回，以确定哪些服务正在请求。执行 SVC 指令期间，当前任务优先级高于或等于 SVC 指令调用处理程序时，将产生一个错误。此指令不影响 N、Z、C、V 状态标志
休眠指令	(56)	WFE	休眠并且在发生事件时被唤醒。不影响 N、Z、C、V 标志
	(57)	WFI	休眠并且在发生中断时被唤醒。不影响 N、Z、C、V 标志

2.3　ARM Cortex-M0+指令集与机器码对应表

ARM Cortex-M0+处理器指令集与机器码对应关系如表 2-19 所示，“机器码”列中，v 代表 immed_value；n 代表 Rn；m 代表 Rm；s 代表 Rs；r 代表 register_list；c 代表 condition；d 代表 Rd；l 代表 label。机器码均为大端对齐方式，即高位字节低地址。

表 2-19　指令集与机器码对应表

分类	序号	助记符	指令格式	机器码	实例	
					指　令	机　器　码
数据传送类指令	1	ADR	ADR　Rd,label	1010 0ddd vvvv vvvv	ADRR3，REPEAT	A305
	2	LDR	LDR　Rd,[RN,RM]	0101 100m mmnn nddd	LDR　r4, [r4, r5]	5964
			LDR　Rd,label	0100 1ddd vvvv vvvv	LDR　r5, =runpin	4D0A
			LDR　Rd,[<RN\|SP>{,#imm}]	0110 1vvv vvnn nddd	LDR　r1, [r5, #0]	6829
	3	LDRH	LDRH　Rd,[Rn{,#imm}]	1000 1vvv vvnn nddd	LDRH　r4, [r4, #30]	8be4
			LDRH　Rd,[Rn,Rm]	0101 101m mmnn nddd	LDRH　r4, [r4, r5]	5b64
	4	LDRB	LDRB　Rd,[Rn{,#imm}]	0111 1vvv vvnn nmmm	LDRB　r4, [r4, #30]	7FA4
			LDRB　Rd,[Rn,Rm]	0101 110m mmnn nddd	LDRB　r4, [r4, r5]	5D64
	5	LDRSH	LDRSH　Rd,[Rn,Rm]V	0101 111m mmnn nddd	LDRSH r4, [r4, r5]	5F64
	6	LDRSB	LDRSB　Rd,[Rn,Rm]	0101 011m mmnn nddd	LDRSB r4, [r4, r5]	5764
	7	LDM	LDM　Rn{!},Reglist	1100 1nnn rrrr rrrr	LDM r0,{r0,r3,r4}	C819
	8	STR	STR Rd, [<RN\|SP>{,#imm}]	0110 0vvv vvnn nddd	STR　r0, [r5, #4]	6068
			STR　Rd,[Rn,Rm]	0101 000m mmnn nddd	STR　r0,[r5,r4]	5128
	9	STRH	STRH　Rd,[Rn{,#imm}]	1000 0vvv vvnn nddd	STRh r0,[r5,#4]	80A8
			STRH　Rd,[Rn,Rm]	0101 001m mmnn nddd	STRh　r0,[r5,r4]	5328
	10	STRB	STRB　Rd,[Rn{,#imm}]	0111 0vvv vvnn nddd	STRB　r0,[r5,#4]	7128
			STRB　Rd,[Rn,Rm]	0101 010m mmnn nddd	STRB　r0,[r5,r4]	5528
	11	STM	STM　Rn!,reglist	1100 0nnn rrrr rrrr	STMIA r0,{r0,r3,r4}	C019

续表

分类	序号	助记符	指令格式	机器码	实例	
					指令	机器码
	12	MOV	MOV Rd,Rm	0001 1100 00nn nddd	MOV r1,r2	1C11
	13	MOVS	MOVS Rd,#imm	0010 0ddd vvvv vvvv	MOVS r1,#8	2108
	14	MVN	MVN Rd,Rm	0100 0011 11mm mddd	MVN r1,r3	43D9
	15	PUSH	PUSH reglist	1011 010R rrrr rrrr	PUSH {r1}	B402
	16	POP	POP reglist	1011 110R rrrr rrrr	PUSH {r1}	BC02
数据操作类指令	17	ADC	ADC {Rd} Rn,Rm	0100 0001 01mm mddd	ADC r2,r3	415A
	18	ADD	ADD {Rd} Rn<Rm\|#imm>	0001 100m mmnn nddd	ADD r2,r3	18D2
			ADD Rn,#imm	0011 0ddd vvvv vvvv	ADD r2,#12	320C
			ADD Rd,Rn,#imm	0001 110v vvnn nddd	ADD r2,r3,#1	1C5A
			ADD Rd,Rn,#imm	0001 100m mmnn nddd	ADD r2,r3,r4	191A
	19	RSB	(KDS 环境不支持)			
	20	SBC	SBC {Rd,} Rn, Rm	0100 0001 10mm mnnn	SBC R7,R7,R1	418F
	21	SUB	SUB Rd,#imm	0011 1ddd vvvv vvvv	SUB r4,#1	3C01
			SUB Rd,Rn,#imm	0001 111v vvnn nddd	SUB r3,r4,#1	1E63
			SUB Rd,Rn,Rm	0001 101m mmnn nddd	SUB r3,r4,r1	1A63
	22	MUL	MUL Rd,Rn,Rm	0100 0011 01mm mddd	MUL r1,r2,r1	4351
	23	CMN	CMN Rn,Rm	0100 0010 11mm mnnn	CMN r1,r2	42D1
	24	CMP	CMP Rn,#imm	0010 1nnn vvvv vvvv	CMP r4,#1	2C01
			CMP Rn,Rm	0100 0010 10mm mnnn	CMP r4,r1	428C
	25	AND	AND {Rd,}Rn,Rm	0100 0000 00mm mddd	AND r1,r2	4011
	26	ORR	ORR {Rd,}Rn,Rm	0100 0011 00mm mddd	ORR r1,r2	4311
	27	EOR	EOR {Rd,}Rn,Rm	0100 0000 01mm mddd	EOR r1,r2	4051
	28	BIC	BIC {Rd,}Rn,Rm	0100 0011 10mm mddd	BIC r1,r2	4391
	29	REV	REV Rd,Rn	1011 1010 00nn nddd	REV r1,r2	BA11
	30	REV16	REV16 Rd,Rn	1011 1010 01nn nddd	REV16 r3,r3	BA5B
	31	REVSH	REVSH Rd,Rn	1011 1010 11nn nddd	REVSH r4,r3	BADC
	32	SXTB	SXTB Rd,Rm	1011 0010 01mm mddd	SXTB r4,r3	B25C
	33	SXTH	SXTH Rd,Rm	1011 0010 00mm mddd	SXTH r4,r3	B21C
	34	UXTB	UXTB Rd,Rm	1011 0010 11mm mddd	UXTB r4,r3	B2DC
	35	UXTH	UXTH Rd,Rm	1011 0010 10mm mddd	UXTH r4,r3	B29C
	36	TST	TST Rn,Rm	0100 0010 00mm mnnn	TST r1,r2	4211
	37	ASR	ASR {Rd,}Rm,Rs	0100 0001 00ss smmm	ASR r3,r3,r5	412B
			ASR {Rd,}Rm,#imm	0001 0vvv vvmm mddd	ASR r7,r5,#6	11AF
	38	LSL	LSL {Rd,}Rm,Rs	0100 0000 10ss smmm	LSL r5,r6	40B5
			LSL {Rd,}Rm, #imm	0000 0vvv vvmm mddd	LSL r5,r6,#6	01B5
	39	LSR	LSR {Rd,}Rm,Rs	0100 0000 11ss sddd	LSR r5,r6	40F5
			LSR {Rd,}Rm, #imm	0000 1vvv vvmm mddd	LSR r5,r6,#6	09B5
	40	ROR	ROR {Rd,}Rm,Rs	0100 0001 11ss sddd	ROR r5,r6	41F5
跳转类指令	41	B	B label	1110 0vvv vvvv vvvv	B repeat	E7F8
			B{cond} label	1110 cccc vvvv vvvv	BNE repeat	D1F8
	42	BL	BL label	32 位指令	BL loop	F7FF FFD2
	43	BX	BX Rm	0100 0111 00mm m000	BX r1	4708
	44	BLX	BLX Rm	0100 0111 10mm m000	BLX r1	4788

续表

分类	序号	助记符	指令格式	机器码	实例	
					指令	机器码
其他指令	45	BKPT	BKPT　#imm	1011 1110 vvvv vvvv	BKPT	BE00
	46	CPSIE	CPSIE　i	1011 0110 0110 0010	CPSIE　i	B662
	47	CPSID	CPSID　i	1011 0110 0111 0010	CPSID　i	B672
	48	DMB	DMB	32 位指令	DMB	F3BF 8F5F
	49	DSB	DSB	32 位指令	DSB	F3BF 8F4F
	50	ISB	ISB	32 位指令	ISB	F3BF 8F6F
	51	MRS	MRS　Rd,spec_reg1	32 位指令	MRS R5, PRIMASK	F3EF 8510
	52	MSR	MSR　spec_reg,Rn	32 位指令	MSR PRIMASK,R5	F385 8810
	53	NOP	NOP		NOP	46C0
	54	SEV	SEV	1011 1111 0100 0000	SEV	BF40
	55	SVC	SVC　#imm	1101 1111 vvvv vvvv	SVC #12	DF0C
	56	WFE	WFE	1011 1111 0010 0000	WFE	BF20
	57	WFI	WFI	1011 1111 0011 0000	WFI	BF30



2.4　GNU 汇编语言的基本语法

能够在 MCU 内直接执行的指令序列是机器语言，用助记符号来表示机器指令便于记忆，这就形成了汇编语言。因此，用汇编语言写成的程序不能直接放入 MCU 的程序存储器中去执行，必须先转为机器语言。把用汇编语言写成的源程序“翻译”成机器语言的工具叫汇编程序或汇编器（Assembler），以下统一称为汇编器。

本书给出的所有样例程序均在 KDS3.0 开发环境下实现，KDS3.0 环境默认使用 GNU 汇编器，汇编语言格式满足 GNU 汇编语法。为了有助于解释涉及的汇编指令，下面将介绍一些汇编语法的基本信息。

2.4.1　汇编语言格式

汇编语言源程序可以用通用的文本编辑、软件编辑，以 ASCII 码形式存盘。具体的汇编器对汇编语言源程序的格式有一定的要求，同时，汇编器除了识别 MCU 的指令系统外，为了能够正确地产生目标代码，以及方便汇编语言的编写，汇编器还提供了一些在汇编时使用的命令、操作符号，在编写汇编程序时，也必须正确使用它们。由于汇编器提供的指令仅是为了更好地做好“翻译”工作，并不产生具体的机器指令，因此这些指令被称为伪指令（Pseudo Instruction）。例如，伪指令告诉编译器：从哪里开始编译，到何处结束，汇编后的程序如何放置等相关信息。当然，这些相关信息必须包含在汇编源程序中，否则编译器就难以编译好源程序，难以生成正确的目标代码。

汇编语言源程序以行为单位进行设计，每一行最多可以包含以下几个部分。

```
标号：操作码　操作数 1，操作数 2……注释
```

1．标号（Labels）

标号可以确定代码当前位置的程序计数器（PC）值，对于标号有下列要求及说明。

（1）如果一个语句有标号，则标号必须书写在汇编语句的开头部分。

（2）标号可以由任何有效字符和冒号组成。有效字符包含以下字符：字母 A～Z、字母 a～z、数字 0～9、下画线“_”、美元符号“$”，但开头的第一个符号不能为数字和$。

（3）汇编器对标号中字母的大小写敏感，但指令不区分大小。

（4）标号长度基本上不受限制，但实际使用时通常不要超过 20 个字符。若希望更多的汇编器能够识别，建议标号（或变量名）的长度小于 8 个字符。

（5）标号后必须带冒号“：”。

（6）一个标号在一个文件（程序）中只能定义一次，否则重复定义，不能通过编译。

（7）一行语句只能有一个标号，汇编器将把当前程序计数器的值赋给该标号。

2．操作码（Opcodes）

操作码包括指令码或伪指令，其中伪指令是指 GNU 汇编器可以识别的伪指令。对于有标号的行，必须用至少一个空格或制表符（TAB）将标号与操作码隔开；对于没有标号的行，不能从第一列开始写操作码，应以空格或制表符（TAB）开头。汇编器不区分操作码中字母的大小写。

3．操作数（Operands）

操作数可以是地址、标号或指令码定义的常数，也可以是由伪运算符构成的表达式。若一条指令或伪指令有操作数，则操作数与操作码之间必须用空格隔开书写。操作数多于一个的，操作数之间用逗号“,”分隔。操作数也可以是 ARM Cortex-M0+内部寄存器，或者另一条指令的特定参数。操作数中一般都有一个存放结果的寄存器，这个寄存器在操作数的最前面。

（1）常数标识。汇编器识别的常数有十进制（默认不需要前缀标识）、十六进制（0x 前缀标识）、二进制（用 0b 前缀标识）。

（2）“#”表示立即数。一个常数前添加“#”表示一个立即数，不加“#”时，表示一个地址。

特别说明：初学者常常会将立即数前的“#”遗漏，如果该操作数只能是立即数时，汇编器会提示错误，例如：

```
mov     r3, 1     @给寄存器 r3 赋值为 1（这个语句不对）
```

编译时会提示“immediate expression requires a # prefix – 'mov r3,1'”，应该改为

```
mov     r3，#1     @给寄存器 r3 赋值为 1（这个语句对）
```

（3）圆点“.”。若圆点“.”单独出现在语句的操作码之后的操作数位置上，则代表当前程序计数器的值被放置在圆点的位置。例如，“b.”指令代表转向本身，相当于永久循环，在调试时希望程序停留在某个地方可以添加这种语句，调试之后应删除。

（4）伪运算符。表 2-20 列出了一系列的伪运算符。

表 2-20　GNU 汇编器识别的伪运算符

	运 算 符	功　　能	类　　型	实　　例	
运算符前缀	-	负号	一元	ldr　r3,=-325	等价于 ldr　r3,=0xfffffebb
	～	取反运算	一元	ldr　r3,=～325	等价于 ldr　r3,=0xfffffeba
后缀运算符，优先级从大到小	*	乘法	二元	mov　r3,#5*4	等价于 mov　r3,#20
	/	除法	二元	mov　r3,#20/4	等价于 mov　r3,#5
	%	取模	二元	mov　r3,#20%7	等价于 mov　r3,#6
	<或<<	左移	二元	mov　r3,#4<<2	等价于 mov　r3,#16
	>或>>	右移	二元	mov　r3,#4>>2	等价于 mov　r3,#1
	\|	按位或	二元	mov　r3,#4\|2	等价于 mov　r3,#6
	&	按位与	二元	mov　r3,#4^2	等价于 mov　r3,#0
	^	按位异或	二元	mov　r3,#4^6	等价于 mov　r3,#2
	!	逻辑非	二元	mov　r3,！#1	等价于 mov　r3,#0
	+	加法	二元	mov　r3,#30+40	等价于 mov　r3,#70
	-	减法	二元	mov　r3,#40-30	等价于 mov　r3,#10
	==	等于	二元	mov　r3,#1==0	等价于 mov　r3,#0
	<>	不等于	二元	mov　r3,#1<>1	等价于 mov　r3,#0
	>	大于	二元	mov　r3,#1>0	等价于 mov　r3,#1
	<	小于	二元	mov　r3,#1<0	等价于 mov　r3,#0
	>=	大于等于	二元	mov　r3,#1>=0	等价于 mov　r3,#1
	<=	小于等于	二元	mov　r3,#1<=0	等价于 mov　r3,#0
	&&	逻辑与	二元	mov　r3,#1&&0	等价于 mov　r3,#0
	\|\|	逻辑或	二元	mov　r3,#1\|\|0	等价于 mov　r3,#1

4．注释（Comments）

注释即说明文字，类似于 C 语言，多行注释以“/*”开始，以“*/”结束。这种注释可以包含多行，也可以独占一行。在 GNU 汇编器的汇编语言中，单行注释以“@”引导或者用“#”引导，用@引导类似于 C 语言的“//”。用“#”引导，“#”必须为单行的第一个字符。

2.4.2　伪指令

为了方便汇编语言的编写，以及编译器能够正确地产生目标代码，编译器还提供了一些伪指令。所谓伪指令，就是没有对应的机器码的指令，它是用于告诉编译器如何进行汇编的指令，它既不控制机器的操作也不被汇编成机器代码，只能为编译器所识别并指导如何进行汇编。这些伪指令主要有用于常量及宏的定义、条件判断、文件包含等伪指令。所有的汇编命令都是以"."开头。

1．系统预定义的段

在 KDS3.0 开发环境下，C 语言程序在经过 gcc 编译器最终生成.elf 格式的可执行文件。.elf 可执行程序是以段为单位来组织文件的，通常划分为.text、.data 和.bss 几个段，其中，.text

是只读的代码区，.data 是可读可写的数据区，而.bss 则是可读可写且没有初始化的数据区。.text 段开始地址为 0x0，接着分别是.data 段和.bss 段。

```
.text           @表明以下代码在.text 段
.data           @表明以下代码在.data 段
.bss            @表明以下代码在.bss 段
```

2．常量的定义

汇编代码常用的功能之一为常量的定义。使用常量定义，能够提高程序代码的可读性，并且使代码维护更加简单。常量的定义可以使用.equ 汇编指令，下面是 GNU 汇编器的一个常量定义的例子：

```
.equ _NVIC_ICER,  0xE000E180
……
LDR    R0,=_NVIC_ICER                @将 0xE000E180 放到 R0 中
```

常量的定义还可以使用.set 汇编指令，其语法结构与.equ 相同。

```
.set ROM_size, 128 * 1024          @ROM 大小为 31 072 字节(128 KB)
.set start_ROM, 0xE0000000
.set end_ROM, start_ROM + ROMsize  @ROM 结束地址为 0xE0020000
```

3．程序中插入常量

对于大多数汇编工具来说，一个典型特性为可以在程序中插入数据。GNU 汇编器语法可以写为

```
    LDR R3,=NUMNER              @得到 NUMNER 的存储地址
    LDR R4,[R3]                 @将 0x123456789 读到 R4
    ……
    LDR R0,=HELLO_TEXT          @得到 HELLO_TEXT 的起始地址
    BL  PrintText               @调用 PrintText 函数显示字符串
    ……
    ALIGN 4
NUMNER:
    .word  0x123456789
HELLO_TEXT:
    .asciz "hello\n"            @以'\0'结束的字符
```

为了在程序中插入不同类型的常量，GNU 汇编器中包含许多不同的伪指令，表 2-21 中列出了常用的例子。

表 2-21　用于程序中插入不同类型常量的常用伪指令

插入数据的类型	GNU 汇编器
字	.word/.4byte（例如，.word 0x123456789）
半字	.hword/.4byte（例如，.word 0x12345）
字节	.byte（例如，.byte 0x12）
字符串	ascii/.asciz（例如，.ascii “hello\n”，.asciz 与.ascii，只是生成的字符串以‘\0’结尾）

4．条件伪指令

.if 条件伪指令后面紧跟着一个恒定的表达式(即该表达式的值为真)，并且最后要以.endif 结尾。中间如果有其他条件，可以用.else 填写汇编语句。

.ifdef 标号，表示如果标号被定义，执行下面的代码。

5．有限循环

一种循环将汇编语句中的标号按照给定的值替换掉，执行相应的次数。该循环以.irp 开始，以.endr 结束。每次循环标号都设置成规定的值进行执行，如果没有值，标号默认为空字符。汇编语句中用到标号的地方用“\标号”代替。

```
.irp param,1,2,3
mov r\param,#4
.endr
```

编译器将上面的伪指令直接转换成下面的汇编语句执行。

```
mov r1,#4
mov r2,#4
mov r3,#4
```

还有一种循环以循环次数为循环执行标准，以.rept 开头，以.endr 结尾。

```
.rept 3
ADD  r1,#4
.endr
```

编译器将上面的伪指令直接转换成下面的汇编语句执行。

```
ADD  r1,#4
ADD  r1,#4
ADD  r1,#4
```

6．文件包含伪指令

```
.include  "filename"
```

.include 是一个附加文件的链接指示命令，利用它可以把另一个源文件插入当前的源文件一起汇编，成为一个完整的源程序。filename 是一个文件名，可以包含文件的绝对路径或相对路径，但建议对于一个工程的相关文件放到同一个文件夹中，所以更多的时候使用相对路径。具体例子参见第 3 章的第一个汇编实例程序。

7．宏定义和宏调用伪指令

宏定义伪指令为：

```
.macro  macname [macargs...]
……
.endm
```

定义一个宏，以.macro 开始，以.endm 结束。如果宏使用参数，那么在宏体中使用该参

数时添加前缀“\”。宏定义时的参数还可以使用默认值。在宏内部表达式中，可以用.exitm 伪指令直接退出宏，放弃后面命令。

例如，定义一个宏执行一段代码，执行一段命令。

```
.macro  sum from, to
    .long   \from
    .if     \to-\from
        sum     "(\from+1)",\to
    .endif
.endm
```

macro 后面的参数是宏被调用时的参数，定义的宏名称不能与汇编指令及汇编伪指令名称相冲突，可以是标识符。调用这个宏 sum 方法如下。

```
sum 0,5
```

这个宏等同于如下程序段。

```
.long   0
.long   1
.long   2
.long   3
.long   4
.long   5
```

8．其他常用伪指令

除了上述的伪指令外，GNU 汇编还有其他常用伪指令。

（1）.section 伪指令。用户可以通过.section 伪指令来自定义一个段。例如：

```
.section .isr_vector, "a"       @定义一个.isr_vector 段，"a"表示允许段
```

（2）.global 伪指令。.global 伪指令可以用来定义一个全局符号。例如：

```
.global  symbol                 @定义一个全局符号 symbol
```

（3）.extern 伪指令。.extern 伪指令的语法为“.extern symbol”，声明 symbol 为外部函数，调用的时候可以遍访所有文件找到该函数并且使用它。例如：

```
.extern  main                   @声明 main 为外部函数
bl main                         @进入 main 函数
```

（4）.align 伪指令。.align 伪指令可以通过添加填充字节使当前位置满足一定的对齐方式。语法结构为“.align alignment[, [fill][, max]]”，其中，fill 为对齐方式（必须是 2 的倍数），max 给出为对齐而填充的字节值，可省略，默认为 0x00。例如：

```
.align  2   @把当前位置计数器值增加到 2 的倍数上，如果已经是 2 的倍数，不做任何改变
```

（5）.print 伪指令。.print 伪指令的语法格式是“.print string”，表示打印 string 到标准输出。

（6）.end 伪指令。.end 伪指令声明汇编文件的结束。

第 3 章

KEA128 存储映像、中断与硬件最小系统

本章导读：本章简要概述了 KEA128 的存储映像、中断系统与硬件最小系统，有助于读者了解 KEA128 软硬件系统的大致框架，以便开始 KEA128 的软硬件设计。本章 3.1 节给出 KEA128 系列芯片的存储映像，包括 Flash 区、片内 RAM 区、外设区、系统保留段等，以便于配置链接文件；3.2 节给出 KEA128 中断系统介绍，包括 KEA128 的中断源、中断向量表及 Cortex-M0+非内核模块中断的编程结构，以助于理解中断系统；3.3 节将 KEA128 的引脚分为硬件最小系统引脚及 I/O 端口资源类引脚两类，并给出其各自的功能介绍；3.4 节给出 KEA128 硬件最小系统的原理图，并对其分块进行介绍，作为硬件设计时的参考。

本章参考资料：本章 3.1 节的 KEA128 系列存储映像与 3.2 节的 KEA128 的中断源参考自《KEA128 参考手册》的第 3 章；3.2.3 节的 Cortex-M0+非内核模块中断及 NVIC 寄存器介绍参考自《ARMv6-M 架构参考手册》；3.3 节的 KEA128 的引脚功能，参考自《KEA128 参考手册》的第 10 章。

3.1 KEA128 系列存储映像

KEA128 把 Cortex-M0+内核之外的模块，用类似存储器编址的方式，统一分配地址。在 4 GB 的存储映像空间内，片内 Flash、SRAM、系统配置寄存器，以及其他外设如 GPIO，被配有独立地址，以便 Cortex-M0+内核进行访问。表 3-1 给出了本书使用的 KEA128（SKEAZ128MLK）存储映像。

表 3-1 KEA128（SKEAZ128MLK）系统存储映像

区 域 划 分	系统 32 位地址范围	功能说明
Flash 区	0x0000_0000～0x07FF_FFFF	Flash 空间，本芯片只使用 0x0000 0000～0x0001 FFFF，128 KB，其中前 192 B 为中断向量表
	0x0800_0000～0x1FFF_EFFF	保留
片内 RAM 区	0x1FFF_F000～0x1FFF_FFFF	SRAM_L 空间，4 KB（普通 RAM 区）
	0x2000_0000～0x2000_2FFF	SRAM_U 空间，12 KB（支持位操作 RAM 区）
	0x2000_3000～0x21FF_FFFF	保留
	0x2200_0000～ 0x2205_FFFF	384 KB（映射到 12 KB 的 SRAM_U 位带别名区）
	0x2206_0000～0x23FF_FFFF	保留
	0x2400_0000～0x3FFF_FFFF	448 MB（位操作引擎 BME 访问 SRAM_U）
外设区	0x4000_0000～0x4007_FFFF	AIPS 外设，如串口、定时器、模块配置等
	0x4008_0000～0x400F_EFFF	保留
	0x400F_F000～0x400F_FFFF	通用输入/输出（GPIO）模块
	0x4010_0000～0x43FF_FFFF	保留
	0x4400_0000～0x5FFF_FFFF	448 MB（位操作引擎 BME 访问外设 0～127 槽）

续表

区 域 划 分	系统 32 位地址范围	功能说明
外部 RAM、外设区	0x6000_0000～0xDFFF_FFFF	保留
私有外设总线区	0xE000_0000～0xE00F_FFFF	私有外设，如系统时钟、中断控制器、调试接口
系统保留区	0xE010_0000～0xF000_1FFF	保留
	0xF000_2000～0xF000_2FFF	ROM 表，存放存储映射信息
	0xF000_3000～0xF000_3FFF	杂项控制单元
	0xF000_4000～0xF7FF_FFFF	保留
	0xF800_0000～0xFFFF_FFFF	IOPORT：GPIO（单周期访问），可被内核直接访问

3.1.1 Flash 区存储映像

Flash 区（0x0000_0000～0x1FFF_EFFF）一般被用来存储中断向量、程序代码、常数数据等，总计 512 MB。在 KEA128 中，实际上只使用了一块 128 KB（0x0000_0000～0x0001_FFFF）作为该芯片 Flash 区，只占用了该区域一小部分空间，其中前 192 B 为中断向量表。

3.1.2 片内 RAM 区存储映像

片内 RAM 区域一般被用来存储全局变量、临时变量（堆栈空间）等。在 Cortex-M0+中，该区域划分为（0x2000_0000 ～0x3FFF_FFFF），但芯片制造商可以重新调整。KEA128 芯片的片内 RAM 区的 SRAM_L 地址范围被重新调整，将 SRAM_L 高地址固定在 0x1FFF_F000，占据该高地址之前的空间，将 SRAM_U 低地址固定在 0x2000_0000，占据该低地址之后的空间。

飞思卡尔公司给出了 KEA128 芯片的 RAM 区划分计算方法，片内 RAM 被划分为 SRAM_L 和 SRAM_U 两个部分[①]，设 SRAM 空间的大小为 N 字节，其地址范围计算公式为

$$\text{SRAM_L 地址范围}=[0x2000_0000-(N/4)]\sim 0x1FFF_FFFF$$

$$\text{SRAM_U 地址范围}=0x2000_0000\sim[0x2000_0000+(3N/4))-1]$$

KEA128 中，SRAM 的大小为 16 KB，分为 SRAM_L：0x1FFF_F000～0x1FFF_FFFF（4KB）；SRAM_U：0x2000_0000～0x2000_2FFF（12 KB）。

3.1.3 外设区存储映像

通过 AHB 总线和 APB 总线所连接的外设的地址都被划分在此段中(0x4000_0000～0x5FFF_FFFF)。

1．AIPS-Lite（外设桥）存储映像

KEA128 芯片分配给外设区地址空间 1 024 KB（0x4000_0000～0x400F_FFFF），划分成三个区域：128 KB 存储区给片内外设；384 KB 存储空间给片外外设；最后 4 KB 存储区的外设槽用于访问 GPIO 模块。外设桥连接的片内外设有 GPIO 控制器；片外外设有 Flash 存储器、PIT、FTM、ADC、RTC、WDOG、ICS、OSC、I2C、UART 等外设模块。

① 将 SRAM 划分为 SRAM_L 和 SRAM_U。SRAM_U 不仅可以作为普通 RAM 来操作，还可以支持两种途径的位操作，分别是位带别名区、位操作引擎（BME），供特殊功能下高级编程使用。

2．通用输入/输出（GPIO）存储映像

KEA128 芯片分配给外设区地址空间 4 KB（0x400F_F000～0x400F_FFFF），该存储空间用于访问 GPIO 模块。

3．BME（位操作引擎）存储映像

支持 BME 位操作引擎存储区地址位于 0x4400_0000～0x5FFF_FFFF，用于对外设的位操作，位操作引擎技术由硬件支持，可使用 Cortex-M 指令集中最基本的加载、存储指令完成对外设地址空间内存的读、改、写操作。

3.1.4　私有外设总线存储映像

PPB（Private Peripheral Bus）私有外设总线是 ARM 总线体系结构的一部分，这条基于 APB 总线协议的 32 位总线，负责访问位于 0xE000_0000～0xE00F_FFFF 区间的私有外设。

3.1.5　系统保留段存储映像

从 0xE010_0000～0xFFFF_FFFF 的区域为系统保留区域，包括杂项控制单元、ROM 表及 IOPORT 模块等。

其中 ROM 表位于 0xF000_2000 处，它只是一个简单的查找表，记录每个模块的入口地址，提供存储器映射信息，这些信息包括了多种系统设备和调试组件。当调试系统定位各调试组件时，它需要找出相关寄存器在存储器的地址（如 NVIC）和各个调试功能块的地址，调试系统自动检测在 KEA128 芯片中包含了哪些调试组件。在检测到了调试组件后，调试器可以接下来查看它们的 ID 寄存器，从而判定系统中哪些组件是可用的。

3.2　KEA128 中断系统及 Cortex-M0+ 非内核模块中断编程结构



3.2.1　KEA128 的中断源

中断是计算机发展中一个重要的技术，它的出现很大程度上解放了处理器，提高了处理器的执行效率。所谓中断，是指 MCU 在正常运行程序时，由于 MCU 内核异常或者 MCU 各模块发出请求事件，使 MCU 停止正在运行的程序，而转去处理异常或执行处理外部事件的程序（又称为中断服务程序）。

这些引起 MCU 中断的事件称为中断源。KEA128 的中断源分为两类如表 3-2 所示，一类是内核中断，另一类是非内核中断。内核中断主要是异常中断，也就是说，当出现错误的时候，这些中断会复位芯片或者做出其他处理。非内核中断是指 MCU 各个模块被中断源引起的中断，MCU 执行完中断服务程序后，又回到刚才正在执行的程序，从停止的位置继续执行后续的指令。非内核中断又称为可屏蔽中断，这类中断可以通过编程控制，开启或关闭该中断。

表 3-2 中还给出了各个中断源的中断向量序号，非内核中断的中断请求（Interrupt Request）号（简称 IRQ 中断号），以及非内核中断的优先级设置的寄存器号（简称 IPR 寄存器号）。中断向量序号是每一个中断源的固定编号，由芯片设计生产时决定的，编程时不能更改，它代表了中断服务程序入口地址在中断向量表的位置。IRQ 中断号是非内核中断源的编号，每一个编号代表一个非内核中断源。

表 3-2 KEA128 的中断源

中断类型	中断向量号	IRQ 中断号	IPR 寄存器号	中 断 源	
内核中断	0～3			ARM 内核	
	4～10			预留	
	11			ARM 内核	
	12～13			预留	
	14～15			ARM 内核	
非内核中断	16～19	0～3	0	预留	
	20	4	1	预留	
	21	5	1	FTMRE	Flash 命令完成和读冲突中断
	22	6	1	PMC	低电压检测和低电压警告中断
	23	7	1	IRQ	外部中断
	24	8	2	I2C0	I2C0 中断
	25	9	2	I2C1	I2C1 中断
	26	10	2	SPI0	SPI0 中断
	27	11	2	SPI1	SPI1 中断
	28	12	3	UART0	UART0 状态和错误中断
	29	13	3	UART1	UART1 状态和错误中断
	30	14	3	UART2	UART2 状态和错误中断
	31	15	3	ADC0	ADC 转换完成中断
	32	16	4	ACMP0	ACMP 中断
	33	17	4	FTM0	FTM0 中断
	34	18	4	FTM1	FTM1 中断
	35	19	4	FTM2	FTM2 中断
	36	20	5	RTC	RTC 溢出中断
	37	21	5	ACMP1	ACMP1 中断
	38	22	5	PIT CH0	PIT CH0 溢出中断
	39	23	5	PIT CH1	PIT CH1 溢出中断
	40	24	6	KBI0	键盘中断 0（32 位）
	41	25	6	KBI1	键盘中断 1（32 位）
	42	26	6	预留	预留中断
	43	27	6	ICS	ICS 中断
	44	28	7	WDOG	看门狗超时中断
	45	29	7	PWT	PWT 中断
	46	30	7	MSCAN	MSCAN 接收中断
	47	31	7	MSCAN	MSCAN 发送中断、错误和唤醒中断

3.2.2 KEA128 中断向量表

中断向量表是指按照中断源的中断向量序号的固定顺序，存放中断服务程序入口地址的一段存储区域。每个中断服务程序入口地址占用 4 个字节单元，KEA128 中断向量表的位置在存储区 0x0000_0000～0x0000_00BF 的一段地址范围，一共 48×4=192 个字节，存放 48 个

中断服务程序的入口地址。中断服务程序的入口地址又称为中断向量或中断向量指针，它指向中断服务程序在存储器中的位置。

如何将中断服务程序的入口地址（中断向量），按照中断源的中断向量序号的顺序，写入到中断向量表的位置呢？其实，实现的方法也很简单，定义一个常量指针数组，再把中断向量存入这个数组中，工程编译时由链接文件 intflash.ld 指定这个数组存放到中断向量表的位置。关于链接文件 intflash.ld 的功能在 4.4 节中说明，这里不再重复。这个常量指针数组是在 startup_SKEAZ1284.S 文件中实现的。

```
    startup_SKEAZ1284.S 文件
    ……
.section .isr_vector, "a"
    .align 2
    .globl __isr_vector
__isr_vector:                           //中断向量表
    .long   __StackTop                  //Top of Stack
    .long   Reset_Handler               //Reset Handler
    .long   NMI_Handler                 //NMI Handler
    .long   HardFault_Handler           //Hard Fault Handler
    .long   0                           //Reserved
    .long   0                           //Reserved
    .long   0                           //Reserved
    .long   0                           //Reserved
    .long   0                           //Reserved
    .long   0                           //Reserved
    .long   0                           //Reserved
    .long   SVC_Handler                 //SVCall Handler
    .long   0                           //Reserved
    .long   0                           //Reserved
    .long   PendSV_Handler              //PendSV Handler
    .long   SysTick_Handler             //SysTick Handler

                                        //External Interrupts
    .long   Reserved16_IRQHandler       //Reserved interrupt
    .long   Reserved17_IRQHandler       //Reserved interrupt
    .long   Reserved18_IRQHandler       //Reserved interrupt
    .long   Reserved19_IRQHandler       //Reserved interrupt
    .long   Reserved20_IRQHandler       //Reserved interrupt
    .long   FTMRE_IRQHandler    //Command complete and read collision
    .long   PMC_IRQHandler      //Low-voltage detect, low-voltage warning
    .long   IRQ_IRQHandler      //External Interrupt
    .long   I2C0_IRQHandler     //I2C0 interrupt
    .long   I2C1_IRQHandler     //I2C1 interrupt
    .long   SPI0_IRQHandler     //SPI0 single interrupt vector for all sources
    .long   SPI1_IRQHandler     //SPI1 single interrupt vector for all sources
    .long   UART0_IRQHandler    //UART0 status and error
```

```
    .long   UART1_IRQHandler    //UART1 status and error
    .long   UART2_IRQHandler    //UART2 status and error
    .long   ADC_IRQHandler      //ADC interrupt
    .long   ACMP0_IRQHandler    //ACMP0 interrupt
    .long   FTM0_IRQHandler     //FTM0 single interrupt vector for all sources
    .long   FTM1_IRQHandler     //FTM1 single interrupt vector for all sources
    .long   FTM2_IRQHandler     //FTM2 single interrupt vector for all sources
    .long   RTC_IRQHandler      //RTC overflow
    .long   ACMP1_IRQHandler    //ACMP1 interrupt
    .long   PIT_CH0_IRQHandler  //PIT CH0 overflow
    .long   PIT_CH1_IRQHandler  //PIT CH1 overflow
    .long   KBI0_IRQHandler     //Keyboard interrupt0(32bit)
    .long   KBI1_IRQHandler     //Keyboard interrupt1(32bit)
    .long   Reserved42_IRQHandler//Reserved interrupt
    .long   ICS_IRQHandler      //ICS interrupt
    .long   WDOG_IRQHandler     //Watchdog timeout
    .long   PWT_IRQHandler      //PWT interrupt
    .long   MSCAN_RX_IRQHandler //MSCAN Rx Interrupt
    .long   MSCAN_TX_IRQHandler //MSCAN Tx, Err and Wake-up interrupt

    .size    __isr_vector, . - __isr_vector
……
    .align1
    .thumb_func
    .weakDefault_Handler
    .typeDefault_Handler, %function
Default_Handler:                //缺省中断服务例程
   b.
    .sizeDefault_Handler, . - Default_Handler

//Macro to define default handlers. Default handler
//will be weak symbol and just dead loops. They can be
//overwritten by other handlers
//.macrodef_irq_handlerhandler_name
//.weak\handler_name
//.set\handler_name, Default_Handler
//.endm
//用弱符号方式定义缺省的中断向量表项
//Exception Handlers
    def_irq_handler    NMI_Handler
    def_irq_handler    HardFault_Handler
    def_irq_handler    SVC_Handler
    def_irq_handler    PendSV_Handler
    def_irq_handler    SysTick_Handler
    def_irq_handler    Reserved16_IRQHandler
    def_irq_handler    Reserved17_IRQHandler
    def_irq_handler    Reserved18_IRQHandler
```

```
    def_irq_handler   Reserved19_IRQHandler
    def_irq_handler   Reserved20_IRQHandler
    def_irq_handler   FTMRE_IRQHandler
    def_irq_handler   PMC_IRQHandler
    def_irq_handler   IRQ_IRQHandler
    def_irq_handler   I2C0_IRQHandler
    def_irq_handler   I2C1_IRQHandler
    def_irq_handler   SPI0_IRQHandler
    def_irq_handler   SPI1_IRQHandler
    def_irq_handler   UART0_IRQHandler
    def_irq_handler   UART1_IRQHandler
    def_irq_handler   UART2_IRQHandler
    def_irq_handler   ADC_IRQHandler
    def_irq_handler   ACMP0_IRQHandler
    def_irq_handler   FTM0_IRQHandler
    def_irq_handler   FTM1_IRQHandler
    def_irq_handler   FTM2_IRQHandler
    def_irq_handler   RTC_IRQHandler
    def_irq_handler   ACMP1_IRQHandler
    def_irq_handler   PIT_CH0_IRQHandler
    def_irq_handler   PIT_CH1_IRQHandler
    def_irq_handler   KBI0_IRQHandler
    def_irq_handler   KBI1_IRQHandler
    def_irq_handler   Reserved42_IRQHandler
    def_irq_handler   ICS_IRQHandler
    def_irq_handler   WDOG_IRQHandler
    def_irq_handler   PWT_IRQHandler
    def_irq_handler   MSCAN_RX_IRQHandler
    def_irq_handler   MSCAN_TX_IRQHandler
    def_irq_handler   DefaultISR

    .end
```

在包含中断向量表的文件 startup_SKEAZ1284.S 中，主要包含以下几个方面的内容。

（1）定义了中断向量表数组__isr_vector，并将其放在链接文件 intflash.ld 中指定的区域.isr_vector。

（2）为中断向量表的所有表项填入缺省值，即以中断向量所对应外设的英文名作为中断处理函数的函数名。例如，在串口 0 模块的中断向量表项里填入 UART0_IRQHandler。

（3）定义了默认的中断处理函数 Default_Handler，其内容为一个永久循环。

（4）以弱符号[①]的方式，将缺省的中断向量表项都定义为默认中断处理函数 Default_Handler。实际使用时，只需在中断服务程序头文件 isr.h 中再定义一个与所需中断向量缺省函数名相同的函数，再将用户编写的具体中断服务程序名指向其即可。例如：

```
    void UART0_IRQHandler(void);
    #define isr_uart0_re  UART0_IRQHandler
```

① 弱符号可被同名强符号覆盖，C 语言中编译器默认函数和初始化了的全局变量为强符号。

其中函数 UART0_IRQHandler 的名称与串口 0 缺省的中断服务程序名称相同，此时编译器默认将其识别为强符号，在编译时会覆盖掉中断向量表中原来以弱符号定义的缺省值。这时再用宏定义的方式，将用户编写的串口 0 中断服务程序 isr_uart0_re 指向此函数，即可实现外部中断到用户中断服务例程的映射。

到此，中断向量表得以实现。

3.2.3 Cortex–M0+非内核模块中断编程结构

1．Cortex-M0+中断结构及中断过程

Cortex-M0+中断结构原理图如图 3-1 所示，由模块中断源、嵌套向量中断控制器（NVIC）和 Cortex-M0+内核组成。其中断过程分为二步，首先，模块中断源向 NVIC 发出中断请求信号。其次，NVIC 对发来的中断信号进行管理，判断该中断是否允许中断，若允许，通过私有外设总线发送给 Cortex-M0+内核，由内核进行中断处理；如果同时有多个中断信号到来，NVIC 根据设定好的中断信号的优先级进行判断，优先级高的中断首先响应，优先级低的中断挂起，压入堆栈保存；如果优先级完全相同的多个中断源同时请求，则先响应 IRQ 中断号较小的，其他的被挂起。例如，当 IRQ#4 的优先级与 IRQ#5 的优先级相等时，IRQ#4 会比 IRQ#5 先得到响应。

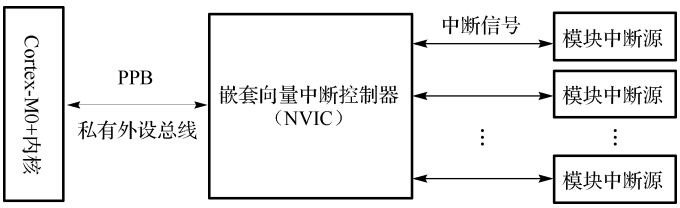


图 3-1 M0+中断结构原理图

2．NVIC 各寄存器简介

NVIC 向量中断控制寄存器共有 12 个寄存器，如表 3-3 所示。下面分别对 NVIC 各寄存器进行介绍。

表 3-3 NVIC 内各寄存器简表

地　址	名　称	描　述
E000_E100	NVIC_ISER	中断使能寄存器（W/R）
E000_E180	NVIC_ICER	中断除能寄存器（W/R）
E000_E200	NVIC_ISPR	中断挂起寄存器（W/R）
E000_E280	NVIC_ICPR	中断清除挂起寄存器（W/R）
E000_E400-E000_E41C	NVIC_IPR0-NVIC_IPR7	优先级寄存器

（1）中断使能寄存器（NVIC_ISER）。中断使能（SET ENABLE）寄存器有 32 位，分别对应 32 个外设 IRQ 中断号。读取第 *n*（0～31）位，如果第 *n* 位为 0，该中断处于禁用状态；如果第 *n* 位为 1，该中断处于使能状态。对第 *n* 位写 1，使能相应 IRQ 号中断，写 0 无效。

（2）中断除能寄存器（NVIC_ICER）。中断除能（CLEAR ENABLE）寄存器有 32 位，分别对应 32 个外设 IRQ 中断号。读取第 *n*（0～31）位，如果第 *n* 位为 0，该中断处于禁用状态；如果第 *n* 位为 1，该中断处于使能状态。对第 *n* 位写 1，禁用相应 IRQ 号中断，写 0 无效。

在编写中断程序中，想要使能一个中断，需要将 NVIC_ISER 中的对应的位置 1；想要对某个中断相应进行除能，需要写 1 到 NVIC_ICER 对应的位。

（3）挂起/清除挂起寄存器（NVIC_ISPR/NVIC_ICPR）。当中断发生时，正在处理同级或者高优先级异常，或者该中断被屏蔽，则中断不能立即得到响应，此时中断被挂起。中断的挂起状态可以通过中断挂起寄存器（NVIC_ISPR）和中断挂起清除寄存器（NVIC_ICPR）来读取，还可以通过写这些寄存器进行挂起中断。这里挂起表示排队等待的意思，清除挂起表示取消此次中断请求。

（4）优先级寄存器（NVIC_IPR0～NVIC_IPR7）。可以通过优先级寄存器设置非内核中断源的优先级。在 KEA128 中优先级寄存器 IPR 共有 8 个（编号 0～7），每一个优先级寄存器对应 4 个非内核中断源。例如，IRQ 中断号为 8、9、10、11 的非内核中断源，它们的优先级都在 IPR 寄存器号为 2 的优先级寄存器中设置。优先级寄存器各字段的含义如下所示，IPR2_IRQ0 字段设置 8 号非内核中断源的优先级，IPR2_IRQ1 字段设置 9 号非内核中断源的优先级，以此类推。因为每个 IPRn_IRQx 字段都由两位组成，所以，非内核中断优先级只能设置为 0～3 级。优先级数值越小，优先级越高。

数据位	31	30	29～24	23	22	21～16	15	14	13～8	7	6	5～0
读	IRQ3		0	IRQ2		0	IRQ1		0	IRQ0		0
写			—			—			—			—

3．中断初始化设置步骤

根据 KEA128 的中断机制，想让一个非内核中断源能够得到内核响应，必须要进行初始化设置。初始化的几个基本步骤如下。

（1）设置模块中断使能位使能模块中断，使模块能够发送中断请求信号。例如，在 UART 中，将控制寄存器 C2 的 RIE 位置 1。

（2）在中断控制器（NVIC）中设置中断使能寄存器（NVIC_ISER），使该模块对应的使能位置 1，允许该中断请求

4．中断初始化示例

串口模块中断初始化函数 uart_enable_re_int()；是包含两个方面的内容，一在控制寄存器 UARTx_C2 中，将接收中断允许位 D5 设置成 1；二是根据 IRQ 号，在断控制器 NVIC 中断使能寄存器（NVIC_ISER），将对应为置 1 开中串口模块中断。程序代码如下。

```
//============================定义串口 IRQ 号对应表=======================
static uint_8 table_irq_uart[3] = {12, 13, 14};
//=========================================================================
//函数名称：uart_enable_re_int
//参数说明：uartNo：串口号:UART_0、UART_1、UART_2
//函数返回：无
//功能概要：开串口接收中断
//=========================================================================
void uart_enable_re_int(uint_8 uartNo)
```

```
{
    UART_MemMapPtr uartch1 = UART_ARR[uartNo];
    uartch1->C2 |= UART_C2_RIE_MASK;            //开放 UART 接收中断
    NVIC_EnableIRQ(table_irq_uart[uartNo]);     //开中断控制器 IRQ 中断
}

//开中断控制器 IRQ 中断
__STATIC_INLINE void NVIC_EnableIRQ(IRQn_Type IRQn)
{
    NVIC->ISER[0] = (1 << ((uint32_t)(IRQn) & 0x1F));
}
```

3.3 KEA128 的引脚功能

本书以 80 引脚 LQFP 封装的 SKEAZ128MLK 芯片为例阐述 ARM Cortex-M0+架构的 Kinetis MCU 的编程和应用。若没有特殊说明，本书的 KEA128 均指 SKEAZ128MLK 芯片。图 3-2 给出的是 80 引脚 LQFP 封装的 SKEAZ128MLK 的引脚图。每个引脚都可能有多个复用功能，有的引脚有两个复用功能，有的有四个复用功能，在实际嵌入式产品的硬件系统设计时必须注意只能使用其中的一个功能。进行硬件最小系统设计时，一般以引脚的第一功能作为引脚名进行原理图设计，若实际使用的是其另一功能，在可以用括号加以标注，这样设计的硬件最小系统就比较通用。

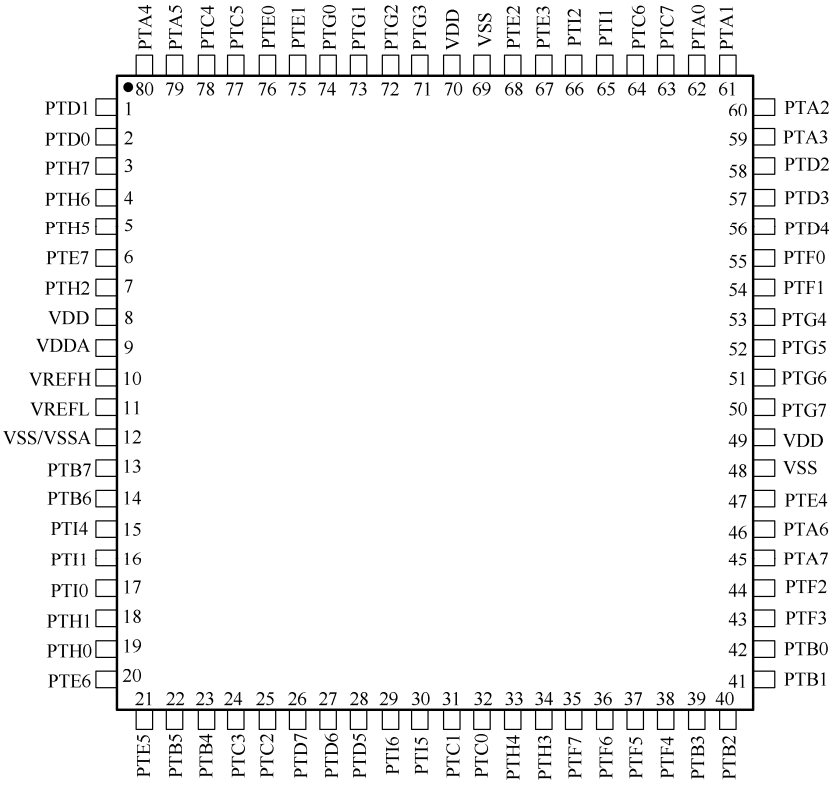


图 3-2　80 引脚 LQFP 封装 SKEAZ128MLK 引脚图

下面从需求与供给的角度把 MCU 的引脚分为硬件最小系统引脚与 I/O 端口资源类引脚两大类。

3.3.1　硬件最小系统引脚

KEA128 硬件最小系统引脚包括电源类引脚、复位引脚、晶振引脚等，如表 3-4 所示。KEA128 芯片电源类引脚，LQFP 封装 9 个。芯片使用多组电源引脚分别为内部电压调节器、I/O 引脚驱动、A/D 转换电路等电路供电，内部电压调节器为内核和振荡器等供电。为了提供稳定的电源，MCU 内部包含多组电源电路，同时给出多处电源引出脚，便于外接滤波电容。为了电源平衡，MCU 提供了内部有共同接地点的多处电源引脚，供电路设计使用。

表 3-4　KEA128 硬件最小系统引脚表

端口名	引脚数	引　脚　名	功 能 描 述
A	8	PTA[0～7]	SWD/FTM/ACMP/ADC/KBI/UART/I2C/RESET/GPIO/IRQ/TCLK
B	8	PTB[0～7]	SPI/FTM/KBI/ADC/UART/ACMP/I2C/NMI/PWT/GPIO
C	8	PTC[0～7]	FTM/ADC/UART/RTC/ACMP/SWD/KBI/CAN/GPIO
D	8	PTD[0～7]	KBI/FTM/SPI/UART/PWT/GPIO
E	8	PTE[0～7]	FTM/SPI/KBI/TCLK/PWT/CAN/GPIO
F	8	PTF[0～7]	KBI/UART/FTM/ADC/GPIO
G	8	PTG[0～7]	KBI/FTM/SPI/GPIO
H	8	PTH[0～7]	KBI/PWT/BUSOUT/FTM/CAN/I2C/GPIO
I	7	PTI[0～6]	IRQ/UART/GPIO
引脚个数统计及说明		LQFP 封装 71 个，这里统计 I/O 引脚不包括已被最小系统使用的引脚。I/O 端口引脚最大输入电压为 $0.7\times V_{DD}$；最大输出电压 V_{DD}，最大输出总电流 100 mA，具体技术指标参见《KEA128 数据手册》	

3.3.2　I/O 端口资源类引脚

除去需要服务的引脚外，其他引脚可以为实际系统提供 I/O 服务。芯片提供服务的引脚也可称为 I/O 端口资源类引脚。KEA128（80 引脚 LQFP 封装）具有 71 个 I/O 引脚。如表 3-5 所示，其中 A 口 8 个、B 口 8 个、C 口 8 个、D 口 8 个、E 口 8 个、F 口 8 个、G 口 8 个、H 口 8 个、I 口 7 个，每个引脚均具有多个功能。这些引脚在复位后，立即被配置为高阻状态，且为通用输入引脚，有内部上拉功能。

表 3-5　KEA128 I/O 端口资源类引脚表

分　　类	引　脚　名	引　脚　号	功 能 描 述
电源输入	VDD	8、49、70	电源，典型值：3.3 V
	VSS	12、48、69	地，典型值：0 V
	VDDA、VSSA	9、12	A/D 模块的输入电源，典型值：（3.3 V、0 V）
	VREFH、VREFL	10、11	A/D 模块的参考电压，典型值：（3.3 V、0 V）

续表

分　类	引 脚 名	引 脚 号	功 能 描 述
复位	RESET	79	复位引脚（双向），该引脚有内部上拉电阻，可在任何模式下通过该引脚完成芯片复位。作为输入，拉低可使芯片复位，拉低脉冲宽度需维持 1.5 个总线时钟周期以上，方能完成复位，作为输出，复位开始后，芯片内部电路驱动该引脚至少维持 34 个总线时钟周期的低电平。上电复位后，该引脚默认为 RESET 功能，复位完成后，可通过系统选项寄存器 SIM_SOPT0 的 RSTPE 位配置为其他功能
晶振	EXTAL、XTAL	13、14	分别为无源晶振输入输出引脚
SWD 接口	SWD_CLK	78	SWD 时钟信号线
	SWD_DIO	80	SWD 数据信号线
引脚个数统计		LQFP 封装 14 个	



3.4 KEA128 硬件最小系统原理图

MCU 的硬件最小系统是指可以使内部程序运行所必需的最小规模的外围电路，也可以包括写入器接口电路。使用一个芯片，必须完全理解其硬件最小系统。当 MCU 工作不正常时，首先就要查找最小系统中可能出错的元件。一般情况下，MCU 的硬件最小系统由电源、晶振及复位等电路组成。芯片要能工作，必须有电源与工作时钟；复位电路则提供不掉电情况下 MCU 重新启动的手段。随着 Flash 存储器制造技术的发展，大部分芯片提供了在板或在线系统（On System）的写入程序功能，即把空白芯片焊接到电路板上后，再通过写入器把程序下载到芯片中。这样，硬件最小系统应该把写入器的接口电路也包含在其中。基于这个思路，KEA128 芯片的硬件最小系统包括电源电路、复位电路、与写入器相连的 SWD 接口电路及可选的晶振电路，具体见附录 B。

3.4.1 电源及其滤波电路

电路中需要大量的电源类引脚用来提供足够的电流容量同时保持芯片电流平衡，所有的电源引脚必须外接适当的滤波电容抑制高频噪音。附录 B 给出了 KEA128 硬件最小系统的电源和地连接图。

电源（VDDx）与地（VSSx）包括很多引脚，如 VDDA、VSSA、VDD、VSS、VREFH 和 VREFL 等。至于外接电容，是由于集成电路制造技术所限，无法在 IC 内部通过光刻的方法制造这些电容。去耦是指对电源采取进一步的滤波措施，去除两级间信号通过电源互相干扰的影响。KEA128 硬件最小系统原理图中，电源滤波电路用于改善系统的电磁兼容性，降低电源波动对系统的影响，增强电路工作的稳定性。为标识系统通电与否，可以增加一个电源指示灯。

3.4.2 复位电路及复位功能

复位电路见 KEA128 硬件最小系统原理图（附录 B）。复位，意味着 MCU 一切重新开始。KEA128 的复位引脚为 PTA5，若 PTA5 信号有效（低电平）则会引起 MCU 复位。PTA5 有一个内部上拉电阻，大小为 30～50 kΩ。复位电路原理为：正常工作时复位输入引脚 PTA5 通过一个内部上拉电阻接到电源正极，所以应为高电平；若按下复位按钮，则 PTA5 脚接地为

低电平，导致芯片复位。KEA128 的复位引脚是双向引脚，作为输入引脚，拉低可使芯片复位，作为输出引脚，上电复位期间有低脉冲输出，表示芯片已经复位完成。

MCU 上电复位（POR）后，系统选项寄存器 0（SIM_SOPT0）的 RESET 引脚使能字段（RSTPE）默认为 1，则 PTA5 引脚复用为复位功能，使能内部上拉电阻。只要不改变 RSTPE 的值，RSTPE 会一直保持为 1。复位脉冲最小宽度为 1.5 个总线时钟周期。在复位电路中，用户也可使用外部上拉电阻，还可附加 1 μF 电容接地及复位按钮。

3.4.3　SWD 接口电路

KEA128 芯片的调试接口 SWD 是基于 ARM CoreSight 架构，该架构在限制输出引脚和其他可用资源情况下，提供了最大的灵活性。CoreSight 是 ARM 定义的一个开放体系结构，以使 SOC 设计人员能够将其他 IP 内核的调试和跟踪功能添加到 CoreSight 基础结构中。通过 SWD 接口可以实现程序下载和调试功能，SWD 接口只需两根线，数据输入/输出线 SWD_DIO 和时钟线 SWD_CLK。附录 B 最小硬件系统原理图中，给出了 SWD 调试接口电路，连接 KEA128 的 PTA4、PTC4 两个引脚，可根据实际需要增加了地、电源以及复位信号线。

3.4.4　晶振电路

KEA128 芯片可使用内部晶振或外部晶振两种方式为 MCU 提供工作时钟。

KEA128 芯片含有内部时钟源（ICS），其频率为 37.5 kHz，误差在 0.8%以内。通过编程，最大可产生 48MHz 内核时钟及 24 MHz 总线时钟。使用内部时钟源可略去外部晶振电路。

若时钟源需要更高的精度，可自行选用外部晶振，图 3-3 给出外接 8 MHz 无源晶振的晶振电路接法，晶振连接在晶振输入引脚 PTB7（EXTAL）、晶振输出引脚 PTB6（XTAL）之间。有关配置及具体编程见第 13.1 节（时钟系统）。

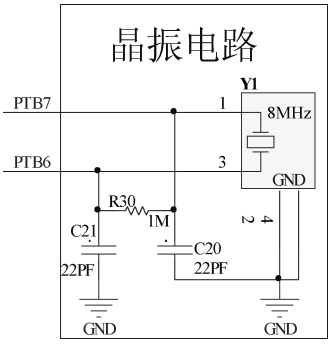


图 3-3　晶振电路

第 4 章

GPIO 及程序框架

本章导读：本章是全书的重点和难点之一，主要目的是通过 GPIO 完成程序框架基本要素设计。本章 4.1 节在简要阐述 GPIO 编程要素基础上，给出 KEA128 芯片 GPIO 引脚标识及技术要点，给出 GPIO 驱动构件接口函数说明及使用方法举例，这样就可以对 KEA128 芯片 GPIO 引脚进行编程操作了；4.2 节给出 GPIO 驱动构件的制作方法，若期望理解 GPIO 驱动构件是如何制作出来的读者，可阅读本节，这一节中端口控制模块功能与编程结构与 GPIO 模块的编程结构总结自《KEA128 数据手册》，随后给出 GPIO 基本打通程序及 GPIO 驱动构件的实现，一般读者可先略过本节，先学会使用再回过头来阅读；4.3 节给出使用 GPIO 驱动构件控制小灯闪烁，目的是给出底层驱动构件的使用范例；4.4 节给出工程框架与第一个 C 语言工程执行过程分析，这是本章重点与难点，期望通过对第一个工程的透彻理解，达到快速、规范入门的目的；若要学习 Cortex-M0+汇编，可阅读 4.5 节。本章内容需配合实际硬件进行阅读，实际硬件只要其 MCU 型号为 KEA128 芯片即可，与具体板无关（可参考工程中的文档说明，设置与板相关性）。

本章参考资料：本章 4.2.1 节（端口控制模块）总结自《KEA128 参考手册》的第 11 章，引脚驱动能力参考自《KEA128 数据手册》的 4.1.1 节；本章 4.2.2 节（GPIO 模块）总结自《KEA128 参考手册》的第 34 章。

4.1 KEA128 芯片 GPIO 驱动构件及使用方法

GPIO（General Purpose Input/Output），俗称通用输入/输出，一般直接读其缩写字母“GPIO”，有的文献也称为并行 I/O，它是输入/输出的最基本形式，其含义是：作为通用输出引脚，MCU 可以通过内部寄存器控制其引脚状态为“1”（高电平）或“0”（低电平），即开关量输出；作为通用输入引脚，MCU 内部程序可以通过其内部寄存器读取该引脚的状态，以确定该引脚是“1”（高电平）或“0”（低电平），即开关量输入[①]。MCU 的大多数 GPIO 引脚可以通过编程来设定其为输入或输出，称为双向通用 I/O。

快速掌握一个芯片 GPIO 的编程，首先需了解芯片 GPIO 引脚分布、命名、引脚特性等基本内容，在此基础上，利用已经制作好的 GPIO 驱动构件进行编程实践。本节给出 KEA128 芯片 GPIO 驱动构件及使用方法，基本满足一般应用需求。若需深入理解 GPIO 驱动构件是如何制作出来的，可参见 4.2 节。

① 这里使用正逻辑表述，简单来说，就是物理上的高电平，对应逻辑数字“1”；低电平，对应逻辑数字“0”。

4.1.1 KEA128 芯片 GPIO 引脚

1．KEA128 芯片 GPIO 引脚的标识

80 引脚 KEA128 芯片共有 71 个引脚可作为 GPIO 引脚，分别为 PORTA、PORTB、PORTC、PORTD、PORTE、PORTF、PORTG、PORTH、PORTI 共 9 个端口。每个端口实际可用的引脚数因封装不同而有差异，80 引脚 KEA128 芯片的 71 个引脚名称标识如下。

- PORTA 口有 8 个引脚，引脚名分别为 PTA7～0；
- PORTB 口有 8 个引脚，引脚名分别为 PTB7～0；
- PORTC 口有 8 个引脚，引脚名分别为 PTC7～0；
- PORTD 口有 8 个引脚，引脚名分别为 PTD7～0；
- PORTE 口有 8 个引脚，引脚名分别为 PTE7～0；
- PORTF 口有 8 个引脚，引脚名分别为 PTF7～0；
- PORTG 口有 8 个引脚，引脚名分别为 PTG7～0；
- PORTH 口有 8 个引脚，引脚名分别为 PTH7～0；
- PORTI 口有 7 个引脚，引脚名分别为 PTI6～0。

2．KEA128 芯片 GPIO 引脚的技术要点

（1）KEA128 引脚基本技术指标[①]。

引脚驱动能力：单个引脚标准驱动能力为 2.5 mA，部分引脚可通过编程设定为高驱动模式，其驱动能力为 10 mA。

内部上下拉电阻：所有数字输入引脚（除了 PTA2 和 PTA3）可配置为内部上拉到 VDD，内部上拉电阻的阻值范围为 30～50 kΩ，PTA2 和 PTA3 为开漏 I/O 引脚[②]（定义为输出时，需要使用外部上拉电阻），KEA128 芯片数字输入引脚无内部下拉。

未接引脚处理方法：建议通过编程将未使用引脚上拉，或外接上拉电阻。

（2）寄存器与引脚的关联。当引脚被配置为 GPIO 功能时，GPIO 数据方向寄存器与输出寄存器控制每个引脚的方向及输出数据；当 GPIO 引脚被定义成输入功能时，输入数据寄存器中一位的逻辑值与对应引脚物理值对应（使用正逻辑）。

（3）引脚复用。许多引脚具有复用功能，当其他复用功能使能时，相应的 GPIO 功能就被关闭。芯片复位后，复用的外设功能会被关闭，引脚自动成为 GPIO 引脚（除了默认被用作 SWD_DIO、SWD_CLK、NMI 和 RESET 功能的 PTA4、PTA5、PTB4 和 PTC4 引脚）。所有的 GPIO 引脚都被配置为 Hi-Z（高阻抗）状态。引脚的复用功能，可以通过系统集成模块（SIM）提供的引脚选择寄存器（Pin Selection Register，SIM_PINSEL）编程来设定使用其中某一种功能[③]。

（4）操作模式。当 MCU 处在运行、等待、调试模式下，GPIO 工作正常；在停止模式下，GPIO 停止工作。

① 摘要自《KEA 数据手册》。

② PTA2 和 PTA3 用于输出时，需要用户接外部上拉电阻。

③ 参见《KEA 参考手册》的 12.2.4 节，引脚复用描述在《KEA 参考手册》的第 10 章。

（5）实时性。GPIO 模块使用系统时钟，从实时性角度来说，当作为通用输出时，高/低电平出现在时钟上升沿。当作为通用输入时，高/低电平可以出现在任何时候，随后并且输入电平在与系统时钟电平异步时生效。

（6）支持零等待状态接口（IOPORT）。KEA128 的 GPIO 模块可以通过系统时钟访问，这是通常编程情况；也支持通过零等待状态接口（IOPORT）与处理器内核通信的编程方式，这种方法支持快速位操作功能[①]。

4.1.2　KEA128 芯片 GPIO 驱动构件头文件及使用方法

在了解 KEA128 芯片 GPIO 引脚基本特性的基础上，本节先行给出 GPIO 驱动构件封装的头文件，目标是读者看完本小节就能够编程，把 GPIO 驱动构件使用起来。

1．GPIO 驱动构件头文件（gpio.h）

```
//=============================================================================
//文件名称：gpio.h
//功能概要：GPIO 底层驱动构件头文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-06-05  V2.0
//芯片类型：KEA128
//=============================================================================

#ifndef GPIO_H          //防止重复定义（GPIO_H  开头）
#define GPIO_H

#include "common.h"     //包含公共要素头文件

//端口号地址偏移量宏定义
#define PORTA    (0<<8)
#define PORTB    (1<<8)
#define PORTC    (2<<8)
#define PORTD    (3<<8)
#define PORTE    (4<<8)
#define PORTF    (5<<8)
#define PORTG    (6<<8)
#define PORTH    (7<<8)
#define PORTI    (8<<8)
//引脚方向宏定义
#define GPIO_IN      0
#define GPIO_OUTPUT  1
//=============================================================================
//函数名称：gpio_init
//函数返回：无
```

① 参见《KEA 参考手册》的 34.3 节，需特殊编程处理。

```
//参数说明：port_pin：(端口号)|(引脚号)（例：PORTB|(5) 表示为 B 口 5 号脚）
//          dir：引脚方向（0=输入，1=输出,可用引脚方向宏定义）
//          state：端口引脚初始状态（0=低电平，1=高电平）
//功能概要：初始化指定端口引脚作为 GPIO 引脚功能，并定义为输入或输出，若是输出，
//          还指定初始状态是低电平或高电平
//=========================================================================
void gpio_init(uint_16 port_pin, uint_8 dir, uint_8 state);
//=========================================================================
//函数名称：gpio_set
//函数返回：无
//参数说明：port_pin：(端口号)|(引脚号)（例：PORTB|(5) 表示为 B 口 5 号脚）
//          state：希望设置的端口引脚状态（0=低电平，1=高电平）
//功能概要：当指定端口引脚被定义为 GPIO 功能且为输出时，本函数设定引脚状态
//=========================================================================
void gpio_set(uint_16 port_pin, uint_8 state);
//=========================================================================
//函数名称：gpio_get
//函数返回：指定端口引脚的状态（1 或 0）
//参数说明：port_pin：(端口号)|(引脚号)（例：PORTB|(5) 表示为 B 口 5 号脚）
//功能概要：当指定端口引脚被定义为 GPIO 功能且为输入时，本函数获取指定引脚状态
//=========================================================================
uint_8 gpio_get(uint_16 port_pin);
//=========================================================================
//函数名称：gpio_reverse
//函数返回：无
//参数说明：port_pin：(端口号)|(引脚号)（例：PORTB|(5) 表示为 B 口 5 号脚）
//功能概要：当指定端口引脚被定义为 GPIO 功能且为输出时，本函数反转引脚状态
//=========================================================================
void gpio_reverse(uint_16 port_pin);
//=========================================================================
//函数名称：gpio_pull
//函数返回：无
//参数说明：port_pin：端口号|引脚号（例：PORTB|(5) 表示为 B 口 5 号脚）
//          pullselect：引脚上拉使能选择（ 0=上拉除能，1=上拉使能）
//功能概要：使指定引脚上拉高电平
//=========================================================================
void gpio_pull(uint_16 port_pin, uint_8 pullselect);
```

2．GPIO 驱动构件的使用方法

在 GPIO 驱动构件的头文件（gpio.h）中包含的内容有：用宏定义方式统一使用的端口名称（PORTA、PORTB、PORTC、PORTD、PORTE、PORTF、PORTG、PORTH、PORTI）；给出 5 个对外服务函数的接口说明及声明，函数包括引脚初始化函数（gpio_init）、设定引脚状态函数（gpio_set）、获取引脚状态函数（gpio_get）、反转引脚状态函数（gpio_reverse）、引脚上拉使能函数（gpio_pull）等。

这里先对引脚的描述方式做一下解释。gpio.h 文件中给出了 GPIO 端口名宏定义，为了

方便使用，期望将“GPIO 引脚”的端口与所在端口逻辑引脚号合在一起进行描述，先把 GPIO 端口名使用数字序号左移 8 位表达，再把端口名与引脚所在端口的逻辑号进行“或”运算，如 PTC0 采用 PORTC|(0)这样的一个 16 位数据描述，其含义是 PORTC 占 16 位数据的高 8 位，低 8 位作为引脚号，数据位数足够描述端口与引脚号，使用 8 位是为了构件更加通用，这样编程时方便使用，构件实现时，也方便从中解析出有关地址。

引脚方向也建议使用宏定义，如把输入 GPIO_IN 定义为 0，输出 GPIO_OUT 定义为 1，见名知意，防止在设置引脚输入/输出时产生混淆。因为我们记不住 0 是输入，还是 1 为输入，所以给出见名知意的宏定义。宏不能乱用，也不能不用，必要时才用。

GPIO 是个特殊构件，它是抽象的，只有具体的物理对象实现时，才有实际使用，如小灯、开关等，4.3 节就使用 GPIO 驱动构件制作 Light 构件。



4.2　GPIO 驱动构件的制作方法

本节给出 KEA128 芯片 GPIO 驱动构件的制作方法，供需要深入理解寄存器编程读者阅读，一般应用开发读者，可以略过本节。

制作 GPIO 驱动构件需要掌握芯片的端口控制（PORT）模块及 GPIO 模块的编程结构，即地址映射、寄存器功能等。

4.2.1　端口控制模块功能与编程结构

端口控制模块（简称 PORT 模块）[①]主要功能为：当引脚为 GPIO 功能时，设置引脚滤波性能、上拉、高驱动能力等。PORT 模块含有端口滤波寄存器、端口上拉使能寄存器、端口高驱动使能寄存器 3 类寄存器。为了更好地理解寄存器功能，下面首先对寄存器描述中出现的相关名词进行解释，随后介绍寄存器功能。

1．相关名词解释

（1）LPOCLK 是 1 kHz 的时钟信号。LPOCLK 在电源管理控制模块 PMC 中的内部低功耗振荡器 LPO 提供的，可为实时时钟 RTC 和看门狗 WDOG 提供参考时钟。

（2）BUSCLK 是总线时钟。由系统时钟分频得到 BUSCLK，此时钟为总线和外设提供时钟源。

（3）模拟引脚（Analog Pin）是指不能够配置成 GPIO 的引脚，如 RESET、EXTAL 及 XTAL 等。KEA128 的所有模拟引脚在芯片内部都有静电阻抗器（Electro-Static Discharge，ESD）保护二极管连接到 VSS 和 VDD。

（4）数字引脚（Digital pin）是指能够被配置成 GPIO 的引脚。所有数字引脚都会通过一个 ESD 保护二极管连接到 VSS。数字引脚分为数字输入（Digital Input）和数字输出（Digital output）两种情况，数字输入是指芯片引脚只能接收并识别高电平（逻辑 1）和低电平（逻辑

① PORT 模块总结自《KEA128 参考手册》的第 11 章。

0）两个电压值；数字输出是指芯片引脚只能输出高电平（逻辑 1）和低电平（逻辑 0）两个电压值。

（5）引脚驱动能力（Drive Strength）是指引脚放出或吸入电流的能力，一般用 mA 作为度量单位。

2．端口滤波寄存器（PORT_IOFLT）

端口滤波寄存器用于为输入引脚设置过滤器，配置高/低干扰阈值。每个数字输入引脚都具有内部滤波功能，可以过滤掉 GPIO、IRQ、RESET、NMI 和 KBI 产生的干扰。干扰阈值可以通过在 1～4 096 个 BUSCLK（或 1～128 个 LPOCLK）内设置 PORT_IOFLT0[FLTDIVn]进行调整。小于选择时钟段长的干扰会被过滤掉，而大于选择时钟段长两倍的不会被过滤掉，并传到内部电路中。KEA128 芯片的端口滤波寄存器 PORT_IOFLTn 共有两个，分别为 PORT_IOFLT0、PORT_IOFLT1，它们均为 32 位寄存器。以下地址分析计算均为十六进制，为书写简化起见，在不至引起歧义的情况下，略去十六进制的前缀“0x”不写。

端口滤波寄存器 PORT_IOFLTn 的基地址=4004_9000，偏移量=4×n，故寄存器 PORT_IOFLTn 的地址=4004_9000h+4×n（n=0～1）。配置寄存器 PORT_IOFLT 可以配置整个端口或者外围输入的滤波器，例如，设置 PORT_IOFLT[FLTA]，会影响所有的 PTAn 的引脚。

（1）端口滤波寄存器 0（PORT_IOFLT0）。

数据位	D31	D30	D29	D28	D27	D26	D25	D24	D23	D22	D21	D20	D19	D18	D17	D16
读/写	FLTDIV3			FLTDIV2			FLTDIV1		FLTNMI		FLTKBI1		FLTKBI0		FLTRST	

数据位	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
读/写	FLTH		FLTG		FLTF		FLTE		FLTD		FLTC		FLTB		FLTA	

下面给出有关功能说明，未说明的位或字段均为保留（只读，值为 0），复位值全为 0。

D31～D29（FLTDIV3）：滤波分频因子 3。这种情况下，f_{LPOCLK} 为分频时钟，设这三位的值记为 n，则滤波时钟频率=$f_{LPOCLK}/2^n$。例如，FLTDIV3=110，选择的时钟为 LPOCLK/64。

D28～D26（FLTDIV2）：滤波分频因子 2。这种情况下，f_{BUSCLK} 为分频时钟，设这三位的值记为 n，则滤波时钟频率=$f_{BUSCLK}/（32\times 2^n）$。例如，FLTDIV2=011，选择的时钟为 BUSCLK/256。

D25～D24（FLTDIV1）：滤波分频因子 1。这种情况下，f_{BUSCLK} 为分频时钟，设这两位的值记为 n，则滤波时钟频率=$f_{BUSCLK}/2^{n+1}$。例如，FLTDIV1=01，选择的时钟为 BUSCLK/4。

D23～D22（FLTNMI）、D21～D20（FLTKBI1）、D19～D18（FLTKBI0）、D17～D16（FLTRST）分别是 NMI、KBI1、KBI0、RESET/IRQ 的滤波方式。设这两位的值记为 n，则 n=0，无滤波；n=1，选择 FLTDIV1 方式，并且，在 Stop 模式下自动转换到 FLTDIV3 方式；n=2，选择 FLTDIV2 方式，并且，在 Stop 模式下自动转换到 FLTDIV3 方式；n=3，选择 FLTDIV3 方式。

D15～D14（FLTH）、D13～D12（FLTG）、D11～D10（FLTF）、D9～D8（FLTE）、D7～D6（FLTD）、D5～D4（FLTC）、D3～D2（FLTB）、D1～D0（FLTA）分别是 PTH、PTG、PTF、PTE、PTD、PTC、PTB、PTA 的滤波方式。设这两位的值记为 n，则 n=0，选择的时

钟为 BUSCLK；*n*=1，选择 FLTDIV1 方式；*n*=2，选择 FLTDIV2 方式；*n*=3，选择 FLTDIV3 方式。

（2）端口滤波寄存器 1（PORT_IOFLT1）。

数据位	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
读	FLTI2C1		FLTI2C0		FLTPWT		FLTFTM1		FLTFTM0		FLTIRQ		0		FLTI	
写													—			

端口滤波寄存器 1 高 16 位未使用，下面给出低 16 位的功能说明，复位值全为 0。

D15～D14（FLTI2C1）、D13～D12（FLTI2C0）、D11～D10（FLTPWT）、D9～D8（FLTFTM1）、D7～D6（FLTFTM0）分别是 SCL1/SDA1、SCL0/SDA0、PWT_IN1/PWT_IN0、FTM1CH0/FTM1CH1、FTM0CH0/FTM0CH1 的滤波方式。设这两位的值记为 *n*，则 *n*=0，选择的时钟为 BUSCLK；*n*=1，选择 FLTDIV1 方式；*n*=2，选择 FLTDIV2 方式；*n*=3，选择 FLTDIV3 方式。

D5～D4（FLTIRQ）：IRQ 的滤波方式。FLTIRQ=00，无滤波；FLTIRQ=01，选择方式为 FLTDIV1，并且，在 Stop 模式下自动转换到 FLTDIV3 方式；FLTIRQ=10，选择方式为 FLTDIV2，并且，在 Stop 模式下自动转换到 FLTDIV3 方式；FLTFTM0=11，选择方式为 FLTDIV3。

D1～D0（FLTI）：PTI 的滤波方式。FLTI=00，选择的时钟为 BUSCLK；FLTI=01，选择方式为 FLTDIV1；FLTI=10，选择方式为 FLTDIV2；FLTI=11，选择方式为 FLTDIV3。

3．端口上拉使能寄存器（PORT_PUE）

端口上拉使能寄存器（PORT_PUE）的功能是：当引脚被配置为数字输入时，设置引脚是否具有内部上拉功能。根据数据手册，KEA128 内部上拉电阻的大小为 30～50 kΩ。若引脚被配置为输出，不能设置内部上拉。KEA128 的引脚只能设置为内部上拉，没有下拉功能。

KEA128 芯片的端口上拉使能寄存器 PORT_PUE 共有三个，分别为 PORT_PUE0、PORT_PUE1、PORT_PUE2，它们均为 32 位寄存器。端口上拉使能寄存器 PORT_PUEn 的基地址=4004_9000h，偏移量=4×（*n*+2），故寄存器 PORT_PUEn 的地址=4004_9000+4×（*n*+2）（*n* =0～2）。

端口上拉使能寄存器 0（PORT_PUE0）的 D31～D24、D23～D16、D15～D8、D7～D0 位分别设置 PTDPE7～PTDPE0、PTCPE7～PTCPE0、PTBPE7～PTBPE0、PTAPE7～PTAPE0 引脚是否具有上拉功能；端口上拉使能寄存器 1（PORT_PUE1）的 D31～D24、D23～D16、D15～D8、D7～D0 位分别设置 PTHPE7～PTHPE0、PTGPE7～PTGPE0、PTFPE7～PTFPE0、PTEPE7～PTEPE0 引脚是否具有上拉功能；端口上拉使能寄存器 2（PORT_PUE2）的 D6～D0 位分别设置 PTIPE6～PTIPE0 引脚是否具有上拉功能。相应位=1，相应引脚上拉使能（Enable）；相应位=0，相应引脚上拉除能（Disable）。

4．端口高驱动使能寄存器（PORT_HDRVE）

电路中某个元器件需要比较高的电流驱动才能工作，这时候需要将引脚驱动力定义为高驱动，所以具有高驱动能力的引脚输出电流可以比一般引脚的高。设置 PORT_HDRVE 寄存

器相应位可以配置数字输出引脚的驱动能力状态。由数据手册，KEA128 引脚正常低驱动电流是 2 mA，高驱动电流是 10 mA，但并不是所有引脚均可配置成高驱动电流。PORT_HDRVE 寄存器的地址为 4004_9014h，具有 32 位，其中，D31～D8 属于保留位，只读，值为 0；D7～D0 分别决定引脚 PTH1、PTH0、PTE1、PTE0、PTD1、PTD0、PTB5、PTB4 的高驱动力能力。相应位=1，相应引脚具有高驱动能力；相应位=0，相应引脚具有一般驱动能力。

4.2.2 GPIO 模块的编程结构

本节端口控制模块的编程结构总结自《KEA128 参考手册》的第 34 章。

从编程角度来说，GPIO 模块有三组 32 位 GPIO 寄存器，分别记为 GPIOA、GPIOB 和 GPIOC。每组包含 7 个寄存器，分别是输出寄存器（GPIOx_PDOR）、输出置 1 寄存器（GPIOx_PSOR）、输出清 0 寄存器（GPIOx_PCOR）、输出反转寄存器（GPIOx_PTOR）、输入寄存器（GPIOx_PDIR）、数据方向寄存器（GPIOx_PDDR）、输入禁止寄存器（GPIOx_PIDR），其中 x=A、B、C。所有 GPIO 引脚由这三组寄存器控制，引脚分组见表 4-1。

表 4-1 GPIO 引脚分组

数据位	D31～D24	D23～D16	D15～D8	D7～D0
GPIOA 组引脚	PTD7～PTD0	PTC7～PTC0	PTB7～PTB0	PTA7～PTA0
GPIOB 组引脚	PTH7～PTH0	PTG7～PTG0	PTF7～PTF0	PTE7～PTE0
GPIOC 组引脚	保留	保留	保留	PTI7～PTI0①

GPIOA、GPIOB 和 GPIOC 的基地址分别为 400F_F000h、400F_F040h、400F_F080h，所以各组基地址相差 40h。其中，输出寄存器的地址就是口的基地址，其他寄存器的地址依次加 4。所有寄存器均为 32 位宽度，每一位对应一个具体的引脚号，例如，GPIOA 组相应寄存器的 D16 位对应 PTC0，决定 PTC0 引脚的控制，复位时均为 0000_0000h（输入禁止寄存器复位时为 FFFF_FFFFh）。表 4-2 给出了 GPIOA 组的 7 个寄存器的基地址、偏移地址、寄存器名、访问特性、功能描述，其他各组功能与编程方式完全一致，只是相应寄存器地址与寄存器名不同。

表 4-2 GPIOA 组寄存器

基 地 址	偏 移	寄 存 器 名	访 问	功 能 描 述
400F_F000h	0h	输出寄存器（GPIOA_PDOR）	R/W	当引脚被配置为输出时，若某一位为 0，对应引脚输出低电平；反之输出高电平
	4h	输出置 1 寄存器（GPIOA_PSOR）	W	写 0 不改变输出寄存器相应位，写 1 将输出寄存器相应位置 1
	8h	输出清 0 寄存器（GPIOA_PCOR）	W	写 0 不改变输出寄存器相应位，写 1 将输出寄存器相应位清 0
	Ch	输出取反寄存器（GPIOA_PTOR）	W	写 0 不改变输出寄存器相应位，写 1 将输出寄存器的相应位取反（即 1 变 0，0 变 1）
	10h	输入寄存器（GPIOA_PDIR）	R	若读出为 0，表明相应引脚上为低电平；若读出为 1，表明相应引脚上为高电平

① 80 引脚 KEA128 芯片只有 PTI6～PTI0

续表

基　地　址	偏　移	寄 存 器 名	访　问	功 能 描 述
	14h	数据方向寄存器（GPIOA_PDDR）	R/W	各位值决定了相对应的引脚为输入还是输出。若其某位设定为 0，则相对应引脚为输入；为 1，则相对应引脚为输出
	18h	输入禁止寄存器（GPIOA_PIDR）	R/W	若某位设定为 0，则配置相应引脚为 GPIO 输入，为 1，引脚不被配置为 GPIO 输入

4.2.3　GPIO 基本打通程序

1．GPIO 基本编程步骤

复位后，引脚默认为 GPIO 功能，首先定义引脚为输入/输出功能，随后进行应用，基本编程步骤如下。

（1）通过 GPIO 模块相应口的“数据方向寄存器”来指定相应引脚为输入或输出功能。若指定位为 0，则为对应引脚输入；若指定位为 1，则为对应引脚输出。

（2）若是输出引脚，则通过设置“输出寄存器”来指定相应引脚输出低电平或高电平，对应值为 0 或 1。亦可通过“输出置位寄存器”、“输出清位寄存器”、“输出取反寄存器”改变引脚状态，参见表 4-2 中关于寄存器的说明。

（3）若是输入引脚，则通过“输入寄存器”获得引脚的状态。若指定位为 0，表示当前该引脚上为低电平；若为 1，则为高电平。

2．理解 GPIO 基本编程步骤举例

举例说明：设 PORTC 口的 0 引脚接一个发光二极管，高电平点亮。现在要点亮这个发光二极管，步骤如下。

（1）确认 PORTC0 所在的寄存器分组，在 GPIOA 组。

（2）通过将数据方向寄存器的 16 位置 1，定义 PORTC0 引脚为输出。

```
GPIOA_PDDR |=(1<<16);
```

（3）通过将输出寄存器的 16 位置 1，使 PORTC0 引脚输出高电平。

```
GPIOA _PDOR|= (1<<16);
```

这样这个发光二极管就亮起来了。该程序样例在本书网上光盘的 “..\02-Software\program\CH04\KEA128-Light-Simple”工程中 main.c 文件中可以看到，用记事本就可打开该文件查看。安装开发环境 KDS，使用方法详见网上光盘中的《KEA128 快速应用指南》，编译该工程，将机器码下载到硬件评估系统中，可以执行该程序。

但是，这样编程只是为了理解 GPIO 的基本编程方法，实际编程并不使用。芯片如此多的引脚，不可能这样编程，要把对底层硬件的访问操作用驱动构件函数封装起来，实际编程时通过访问函数名，给出入口参数。下节将阐述底层驱动构件封装方法与基本规范。

4.2.4　GPIO 驱动构件封装要点分析

本节以 GPIO 驱动构件为例进行封装要点分析，即分析应该设计哪些函数及入口参数。GPIO 引脚可以被定义成输入、输出两种情况：若是输入，程序需要获得引脚的状态（逻辑 1

或 0)；若是输出，程序可以设置引脚状态（逻辑 1 或 0)。MCU 的 PORT 模块分为许多端口，每个端口有若干引脚。GPIO 驱动构件可以实现对所有 GPIO 的引脚统一编程，GPIO 驱动构件由 gpio.h、gpio.c 两个文件组成，如要使用 GPIO 驱动构件，只需要将这两个文件加入到所建工程中，由此方便了对 GPIO 的编程操作。

另外，控制指示灯闪烁的程序中使用了 GPIO 驱动构件。指示灯定义为“端口号|引脚号”，这样便于用户理解和运用。现在添加一个内部解析函数 gpio_port_pin_resolution，将指示灯的定义进行解析，得出具体端口号与引脚号。

1．内部解析函数(gpio_port_pin_resolution)

指示灯定义为端口号|引脚号，例如，PORTB|5 表示为 B 口 5 号引脚。内部解析函数将“端口号|引脚号”解析为具体的端口号和引脚号，以便 GPIO 模块函数使用。

```
static void gpio_port_pin_resolution(uint_16 port_pin,uint_8* port,uint_8* pin)
```

其中，uint_8 是无符号 8 位整型的别名，uint_16 是无符号 16 位整型的别名，其定义在工程文件夹下的“..\ Common”文件夹的 common.h 文件中，本书后面不再特别说明。

2．模块初始化（gpio_init）

由于引脚具有复用特性，把对应引脚设置成 GPIO 功能；同时定义成输入或输出；若是输出，还要给出初始状态。所以 GPIO 模块初始化函数的参数为“端口号|引脚号”、是输入还是输出、若是输出其状态是什么，函数不必有返回值。这样 GPIO 模块初始化函数原型可以设计为

```
void gpio_init(uint_16 port_pin, uint_8 dir, uint_8 state)
```

3．设置引脚状态（gpio_set）

对于输出，希望通过函数设置引脚是高电平（逻辑 1）还是低电平（逻辑 0)，入口参数应该是“端口号|引脚号”，函数不必有返回值。这样设置引脚状态的函数原型可以设计为

```
void gpio_set(uint_16 port_pin, uint_8 state)
```

4．获得引脚状态（gpio_get）

对于输入，希望通过函数获得引脚的状态是高电平（逻辑 1）还是低电平（逻辑 0)，入口参数应该是“端口号|引脚号”，函数需要返回值引脚状态。这样设置引脚状态的函数原型可以设计为

```
uint_8 gpio_get(uint_16 port_pin)
```

5．引脚状态反转（gpio_reverse）

类似的分析，可以设计引脚状态反转函数的原型为

```
void gpio_reverse(uint_16 port_pin)
```

6．引脚上拉使能函数（gpio_pull）

若引脚被设置成输入，还可以设定内部上拉，KEA128 内部上拉电阻大小为 30～50 kΩ。引脚上拉使能函数的原型为

```
void gpio_pull(uint_16 port_pin, uint_8 pullselect)
```

4.2.5　GPIO 驱动构件的实现

根据构件生产的基本要求制作的第一个构件——GPIO 驱动构件，存放于网上光盘“..\KEA128-Light\05_Driver\gpio”文件夹，供复制使用，各个工程文件夹下的“..\05_Driver\gpio”文件夹中 GPIO 驱动构件与此一致。希望仔细分析本构件的编程与注释风格，一开始就规范起来，这样就会逐步锻炼起良好的编程习惯。特别是，不要编写令人难以看懂的程序，不要把简单问题复杂化，不要使用不必要的宏。

1．制作 GPIO 驱动构件使用基地址及结构体类型

KEA128 的端口（PORT）模块基地址为 PORT_BASE_PTR，在芯片头文件中以宏常数方式给出。KEA128 的 GPIO 模块各口基地址为 GPIOA_BASE_PTR、GPIOB_BASE_PTR、GPIOC_BASE_PTR，也在芯片头文件中以宏常数方式给出，本程序直接作为指针常量。

两个结构体类型：端口控制模块结构体类型（PORT_MemMap）与 GPIO 模块结构体类型（GPIO_MemMap）。

在工程文件夹的芯片头文件（“..\MCU\SKEAZ1284.h”）中，有端口寄存器结构体，把端口模块的编程寄存器用一个 PORT 结构体类型（PORT_MemMap）封装起来。

```
typedef struct PORT_MemMap {
 uint32_t  IOFLT0;      //端口滤波寄存器 0（0～31），偏移：0x0，间隔：0x4
 uint32_t  IOFLT1;      //端口滤波寄存器 1（0～31），偏移：0x4
 uint32_t  PUE0;        //端口上拉使能寄存器 0（0～31），偏移：0x8
 uint32_t PUE1;         //端口上拉使能寄存器 1（0～31），偏移：0xC
 uint32_t  PUE2;        //端口上拉使能寄存器 1（0～31），偏移：0x10
 uint32_t  HDRVE;       //端口高驱动使能寄存器，偏移：0x14
} volatile *PORT_MemMapPtr;
```

定义的不优化的 PORT 模块寄存器结构体指针（PORT_MemMapPtr），只要给出端口基地址，就可以使用该结构体的成员变量，实现对各寄存器的访问。类似地，给出了 GPIO 模块结构体类型（GPIO_MemMap）及其指针。

```
typedefstructGPIO_MemMap {
 uint32_t PDOR;         //数据输出寄存器，偏移：0x0
 uint32_t PSOR;         //输出置 1 寄存器，偏移：0x4
 uint32_t PCOR;         //输出清 0 寄存器，偏移：0x8
 uint32_t PTOR;         //输出取反寄存器，偏移：0xC
 uint32_t PDIR;         //数据输入寄存器，偏移：0x10
 uint32_t PDDR;         //数据方向寄存器，偏移：0x14
uint32_t PIDR;          //输入禁止寄存器，偏移：0x18
} volatile *GPIO_MemMapPtr;
```

2．GPIO 驱动构件源程序文件（gpio.c）

GPIO 驱动构件的源程序文件中实现的对外接口函数，主要是对相关寄存器进行配置，从而完成构件的基本功能，构件内部使用的函数也在构件源程序文件中定义。

```
//=============================================================================
//文件名称：gpio.c
//功能概要：GPIO 底层驱动构件源文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-06-05  V2.0
//芯片类型：KEA128
//=============================================================================
#include "gpio.h"       //包含本构件头文件

uint_32 bit;            //内部变量，用于记录引脚在端口寄存器中的偏移量
//内部函数声明
static void gpio_port_pin_num(uint_16 port_pin,uint_8* port,uint_8* pin);

//=============================================================================
//函数名称：gpio_init
//函数返回：无
//参数说明：port_pin：(端口号)|(引脚号)（例：PORTB|(5) 表示为 B 口 5 号脚）
//          dir：引脚方向（0=输入，1=输出,可用引脚方向宏定义）
//          state：端口引脚初始状态（0=低电平，1=高电平）
//功能概要：初始化指定端口引脚作为 GPIO 引脚功能，并定义为输入或输出，若是输出，
//          还指定初始状态是低电平或高电平
//=============================================================================
void gpio_init(uint_16 port_pin, uint_8 dir, uint_8 state)
{
    //局部变量声明
    GPIO_MemMapPtr gpio_ptr;        //声明 port_ptr 为 GPIO 结构体类型指针
    uint_8 port;                    //端口号
    uint_8 pin;                     //引脚号
    gpio_port_pin_num(port_pin, &port, &pin);  //解析出端口号及引脚号
    //计算引脚在寄存器中的偏移量
    if (port < 4)                   //端口号为 PORTA～PORTD
    {
        //端口所在的寄存器基地址
        gpio_ptr = GPIOA_BASE_PTR;
        //引脚所在的位数
        bit = 8 * port + pin;
    }
    else if(3<port &&port< 8)       //端口号为 PORTE～PORTH
    {
        //端口所在的寄存器基地址
        gpio_ptr = GPIOB_BASE_PTR;
        //引脚所在的位数
        bit = 8 * (port - 4) + pin;
```

```
    }
    else                                //端口号为 PORTI
    {
        //端口所在的寄存器基地址
        gpio_ptr = GPIOC_BASE_PTR;
        //引脚所在的位数
        bit = 8 * (port - 8) + pin;
    }
    //根据带入参数 dir，决定引脚为输出还是输入
    if (1 == dir)                       //希望为输出
    {
        //数据方向寄存器定义为输出
        BSET(bit, GPIO_PDDR_REG(gpio_ptr));

        BSET(bit, GPIO_PCOR_REG(gpio_ptr));
        //初始状态为低电平
        gpio_set(port_pin, state);  //调用 gpio_set 函数，设定引脚初始状态
    }
    else
    {
        //希望为输入
        BCLR(bit, GPIO_PDDR_REG(gpio_ptr));
    }
}

//=========================================================================
//函数名称：gpio_set
//函数返回：无
//参数说明：port_pin：端口号|引脚号（例：PORTB|(5) 表示为 B 口 5 号脚）
//          state：引脚初始状态（0=低电平，1=高电平）
//功能概要：设定引脚状态为低电平或高电平
//=========================================================================
void gpio_set(uint_16 port_pin, uint_8 state)
{
    //局部变量声明
    GPIO_MemMapPtr gpio_ptr;    //声明 port_ptr 为 GPIO 结构体类型指针
    uint_8 port;                //端口号
    uint_8 pin;                 //引脚号
    gpio_port_pin_num(port_pin, &port, &pin);  //解析出端口号及引脚号
    //计算引脚在寄存器中的偏移量
    if (port < 4)               //端口号为 PORTA～PORTD
    {
        //端口所在的寄存器基地址
        gpio_ptr = GPIOA_BASE_PTR;
        //引脚所在的位数
        bit = 8 * port + pin;
    }
```

```
    else if(3<port &&port< 8)   //端口号为 PORTE～PORTH
    {
        //端口所在的寄存器基地址
        gpio_ptr = GPIOB_BASE_PTR;
        //引脚所在的位数
        bit = 8 * (port - 4) + pin;
    }
    else                        //端口号为 PORTI
    {
        //端口所在的寄存器基地址
        gpio_ptr = GPIOC_BASE_PTR;
        //引脚所在的位数
        bit = 8 * (port - 8) + pin;
    }

    //根据带入参数 state，决定引脚为输出 1 还是 0
    if (1==state) {BSET(bit,gpio_ptr->PDOR);} else {BCLR(bit,gpio_ptr->PDOR);}
}

//=========================================================================
//函数名称：gpio_get
//函数返回：指定引脚的状态（1 或 0）
//参数说明：port_pin：端口号|引脚号（例：PORTB|(5) 表示为 B 口 5 号脚）
//功能概要：获取指定引脚的状态（1 或 0）
//=========================================================================
uint_8 gpio_get(uint_16 port_pin)
{
    //局部变量声明
    GPIO_MemMapPtr gpio_ptr;        //声明 port_ptr 为 GPIO 结构体类型指针
    uint_8 port;                    //端口号
    uint_8 pin;                     //引脚号
    gpio_port_pin_num(port_pin, &port, &pin);   //解析出端口号及引脚号
    //计算引脚在寄存器中的偏移量
    if (port < 4)                   //端口号为 PORTA～PORTD
    {
        //端口所在的寄存器基地址
        gpio_ptr = GPIOA_BASE_PTR;
        //引脚所在的位数
        bit = 8 * port + pin;
    }
    else if(3<port &&port< 8)       //端口号为 PORTE～PORTH
    {
        //端口所在的寄存器基地址
        gpio_ptr = GPIOB_BASE_PTR;
        //引脚所在的位数
        bit = 8 * (port - 4) + pin;
    }
```

```
    else                                //端口号为 PORTI
    {
        //端口所在的寄存器基地址
        gpio_ptr = GPIOC_BASE_PTR;
        //引脚所在的位数
        bit = 8 * (port - 8) + pin;
    }

    //返回引脚的状态
    return ((BGET(bit,gpio_ptr->PDIR))>=1 ? 1:0);
}

//=========================================================================
//函数名称：gpio_reverse
//函数返回：无
//参数说明：port_pin：端口号|引脚号（例：PORTB|(5) 表示为 B 口 5 号脚）
//功能概要：反转指定引脚输出状态。
//=========================================================================
void gpio_reverse(uint_16 port_pin)
{
    //局部变量声明
    GPIO_MemMapPtr gpio_ptr;        //声明 port_ptr 为 GPIO 结构体类型指针
    uint_8 port;                    //端口号
    uint_8 pin;                     //引脚号
    gpio_port_pin_num(port_pin, &port, &pin);   //解析出端口号及引脚号
    //计算引脚在寄存器中的偏移量
    if (port < 4)                   //端口号为 PORTA～PORTD
    {
        //端口所在的寄存器基地址
        gpio_ptr = GPIOA_BASE_PTR;
        //引脚所在的位数
        bit = 8 * port + pin;
    }
    else if(3<port &&port< 8)       //端口号为 PORTE～PORTH
    {
        //端口所在的寄存器基地址
        gpio_ptr = GPIOB_BASE_PTR;
        //引脚所在的位数
        bit = 8 * (port - 4) + pin;
    }
    else                            //端口号为 PORTI
    {
        //端口所在的寄存器基地址
        gpio_ptr = GPIOC_BASE_PTR;
        //引脚所在的位数
        bit = 8 * (port - 8) + pin;
    }
```

```
    //反转指定引脚输出状态
    BSET(bit,gpio_ptr->PTOR);
}

//=========================================================================
//函数名称：gpio_pull
//函数返回：无
//参数说明：port_pin：端口号|引脚号（例：PORTB|(5) 表示为 B 口 5 号脚）
//         pullselect：引脚上拉使能选择（0=上拉除能，1=上拉使能）
//功能概要：使指定引脚上拉高电平
//=========================================================================
void gpio_pull(uint_16 port_pin, uint_8 pullselect)
{
    //局部变量声明
    uint_8 port;                    //端口号
    uint_8 pin;                     //引脚号
    gpio_port_pin_num(port_pin, &port, &pin);   //解析出端口号及引脚号
    //计算引脚在寄存器中的偏移量
    if (port < 4)                   //端口号为 PORTA～PORTD
    {
        //引脚所在的位数
        bit = 8 * port + pin;
        if (1==pullselect){BSET(bit,PORT_PUE0);}
            else {BCLR(bit,PORT_PUE0);}
    }
    else if(3<port &&port< 8)       //端口号为 PORTE～PORTH
    {
        //引脚所在的位数
        bit = 8 * (port - 4) + pin;
        if (1==pullselect){BSET(bit,PORT_PUE1);}
        else {BCLR(bit,PORT_PUE1);}
    }
    else                            //端口号为 PORTI
    {
        //引脚所在的位数
        bit = 8 * (port - 8) + pin;
        if (1==pullselect){BSET(bit,PORT_PUE2);}
            else {BCLR(bit,PORT_PUE2);}
    }
}

//----------------------以下为内部函数存放处--------------------------------
//=========================================================================
//函数名称：gpio_port_pin_num
//函数返回：无
//参数说明：port_pin：端口号|引脚号（例：PORTB|(5) 表示为 B 口 5 号脚）
//         port：解析出的端口号
```

```
//          pin:解析出的引脚号（0～8，实际取值由芯片的物理引脚决定）
//功能概要：将传进参数 port_pin（例：PORTB|(5)）进行解析，得出具体端口号与引脚号，
//          由 port、pin 传出。
//          例如，PORTB|(5)解析为*port=PORTB，*pin=5)。
//备注：port,pin 为传地址参数，目的是将结果带回，调用函数前，不需要赋值
//=========================================================================
static void gpio_port_pin_num(uint_16 port_pin,uint_8* port,uint_8* pin)
{
    *port = port_pin>>8;
    *pin = port_pin;
}
//----------------------------内部函数结束---------------------------------
```

4.3　第一个 C 语言工程：控制小灯闪烁

本书用 KEA128 控制发光二极管指示灯的例子开始程序之旅。在本书的工程实例中，硬件评估使用 TRK-KEA128 硬件板，该板上有 4 个小灯，灯的正端引脚接 KEA128 的 GPIO 口，负端引脚通过电阻接地，即高电平控制小灯亮，低电平控制小灯暗。TRK-KEA128 硬件板的 4 个小灯，接在 PTC0～PTC3 引脚上。按照 4.1.2 节的 GPIO 驱动构件的使用方法，我们首先使用 GPIO 驱动构件制作 Light 应用构件。

1．Light 构件的头文件 light.h

```
//=========================================================================
//文件名称：light.h
//功能概要：小灯构件头文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2014-12-10  V1.0
//芯片类型：KEA128
//=========================================================================

#ifndef _LIGHT_H                          //防止重复定义（_LIGHT_H  开头)
#define _LIGHT_H

//头文件包含
#include "common.h"                       //包含公共要素头文件
#include "gpio.h"                         //用到 gpio 构件

//指示灯端口及引脚定义
#define LIGHT_0      (PORTC|0)            //灯 0 使用的端口/引脚
#define LIGHT_1      (PORTC|1)            //灯 1 使用的端口/引脚
#define LIGHT_2      (PORTC|2)            //灯 2 使用的端口/引脚
#define LIGHT_3      (PORTC|3)            //灯 3 使用的端口/引脚

//灯状态宏定义（灯亮、灯暗对应的物理电平由硬件接法决定）
#define LIGHT_ON       1        //灯亮
```

```
#define LIGHT_OFF       0        //灯暗


//=================接口函数声明=========================================
//=====================================================================
//函数名称：light_init
//函数参数：port_pin：(端口号)|(引脚号)（如：(PORTB)|(5) 表示为 B 口 5 号脚）
//          state：设定小灯状态。由宏定义。
//函数返回：无
//功能概要：指示灯驱动初始化。
//=====================================================================
void light_init(uint_16 port_pin, uint_8 state);

//=====================================================================
//函数名称：light_control
//函数参数：port_pin：(端口号)|(引脚号)（如：(PORTB)|(5) 表示为 B 口 5 号脚）
//          state：设定小灯状态。由宏定义。
//函数返回：无
//功能概要：控制指示灯亮暗。
//=====================================================================
void light_control(uint_16 port_pin, uint_8 state);

//=====================================================================
//函数名称：light_change
//函数参数：port_pin：(端口号)|(引脚号)（如：(PORTB)|(5) 表示为 B 口 5 号脚）
//函数返回：无
//功能概要：切换指示灯亮暗。
//=====================================================================
void light_change(uint_16 port_pin);

#endif              //防止重复定义（_LIGHT_H  结尾)
```

2．Light 构件的使用方法

现在，以控制一盏小灯闪烁为例，必须知道两点：一是芯片的引脚，二是高电平点亮还是低电平点亮。这样就可使用 Light 构件控制小灯了。例如，小灯由 PTC0 引脚控制，高电平点亮，使用步骤如下。

（1）在 light.h 文件中给小灯起名字，并确定与 MCU 连接的引脚，进行宏定义。

```
#define RUN_LIGHT_0      (PORTC|0)      //灯 0 使用的端口/引脚
```

（2）在 light.h 文件中对小灯亮、暗进行宏定义，方便编程。

```
#define LIGHT_ON    1                   //灯亮
#define LIGHT_OFF   0                   //灯暗
```

（3）在 main 函数中初始化 LED 灯的初始状态。

```
light_init(RUN_LIGHT_0, LIGHT_OFF);    //初始化灯 0
```

（4）在 main 函数中点亮小灯。

```
light_change(RUN_LIGHT_0);              //灯 0（RUN_LIGHT_0）状态变化
```

3．Light 构件的程序文件 light.c 的实现

```
//=============================================================================
//文件名称：light.c
//功能概要：小灯构件源文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2014-12-10  V1.0
//芯片类型：KEA128
//=============================================================================

#include "light.h"

//=============================================================================
//函数名称：light_init
//函数参数：port_pin：(端口号)|(引脚号)（如(PORTB)|(5) 表示为 B 口 5 号脚）
//          state：设定小灯状态，由 light.h 中宏定义
//函数返回：无
//功能概要：指示灯驱动初始化
//=============================================================================
void light_init(uint_16 port_pin, uint_8 state)
{
    gpio_init(port_pin, 1, state);
}

//=============================================================================
//函数名称：light_control
//函数参数：port_pin：(端口号)|(引脚号)（如(PORTB)|(5) 表示为 B 口 5 号脚）
//          state：设定小灯状态，由 light.h 中宏定义
//函数返回：无
//功能概要：控制指示灯亮暗
//=============================================================================
void light_control(uint_16 port_pin, uint_8 state)
{
    gpio_set(port_pin, state);
}

//=============================================================================
//函数名称：light_change
//函数参数：port_pin：(端口号)|(引脚号)（如(PORTB)|(5) 表示为 B 口 5 号脚）
//函数返回：无
//功能概要：切换指示灯亮暗
//=============================================================================
void light_change(uint_16 port_pin)
{
```

```
    gpio_reverse(port_pin);
}
```

4．Light 构件测试工程主程序

测试工程位于网上光盘中的“..\KEA128-Light”文件夹。功能是 KEA128 板上的 Light_0、Light_1、Light_2、Light_3 闪烁。测试工程主程序如下。

```
//说明见工程文件夹下的 Doc 文件夹内 Readme.txt 文件
#define GLOBLE_VAR      //只需在 main.c 中定义一次，用来防止全局变量的重复定义

#include "includes.h"

int main(void)
 {
    //1.声明主函数使用的变量
    uint_32 mRuncount;                    //主循环计数器
    //2. 关总中断
    DISABLE_INTERRUPTS;                   //关总中断
    //3. 初始化外设模块
    light_init(LIGHT_0, LIGHT_OFF);       //初始化灯 0
    light_init(LIGHT_1, LIGHT_OFF);       //初始化灯 1
    light_init(LIGHT_2, LIGHT_OFF);       //初始化灯 2
    light_init(LIGHT_3, LIGHT_OFF);       //初始化灯 3
    //4. main 中使用的局部变量初始化
    mRuncount = 0;                        //主循环计数器
    //5. 全局变量初始化
    //6. 使能模块中断
    //7. 开总中断
    ENABLE_INTERRUPTS;                    //开总中断

    //进入主循环
    //=========================主循环开始=====================================
    for (;;)
    {
        //----------------------运行指示灯闪烁------------------------------
        mRuncount++;                          //主循环次数计数器+1
        if (mRuncount >= RUN_COUNTER_MAX)     //主循环次数计数器大于设定的宏常数
        {
            mRuncount = 0;                    //主循环次数计数器清零
            light_change(LIGHT_0);            //灯 0（LIGHT_0）状态变化
            light_change(LIGHT_1);            //灯 1（LIGHT_1）状态变化
            light_change(LIGHT_2);            //灯 2（LIGHT_2）状态变化
            light_change(LIGHT_3);            //灯 3（LIGHT_3）状态变化
        }
        //---------------------------以下加入用户程序-----------------------

    }  //主循环 end_for
```

```
    //=========================主循环结束=================================
    return 0;
}
```

其中的常数 RUN_COUNTER_MAX 在总头文件中宏定义，决定了小灯闪烁频率。这个程序结构已经很清晰了，也十分容易理解，接下来下载运行。但需要 KDS 开发环境，使用方法参见详见网上光盘中的《KEA128 快速应用指南》。



4.4　工程框架与第一个 C 语言工程执行过程分析

4.4.1　工程框架

图 4-1 给出以 KEA128-Light 工程为例的树形工程结构模板，物理组织与逻辑组织一致。该模板是苏州大学飞思卡尔嵌入式中心为在 KDS 环境下开发 ARM Cortex-M4/M0+ Kinetis K/L 系列 MCU 应用工程而制作的。

KEA128_Light	工程名
Binaries	
Includes	
01_Doc	文档文件夹
02_CPU	内核文件
03_MCU	MCU 相关文件
SKEAZ1284.h	KEAZ128 芯片头文件
startup_SKEAZ1284.S	启动代码
system_SKEAZ1284.c	系统初始化源文件
system_SKEAZ1284.h	系统初始化头文件
04_Linker_File	链接文件夹
intflash.ld	链接文件
05_Driver	驱动文件夹
gpio	GPIO 底层构件文件夹
gpio.c	GPIO 底层构件源文件
gpio.h	GPIO 底层构件头文件
06_App_Component	应用构件文件夹
light	小灯构件文件夹
light.c	小灯构件源文件
light.h	小灯构件头文件
07_Soft_Component	软件构件文件夹
08_Sources	源文件夹
includes.h	总头文件
isr.c	中断源文件
isr.h	中断头文件
main.c	主函数
Debug	工程输出

图 4-1　KEA128-Light 工程的树形工程结构模板

该工程模板，与 KDS3.0 提供的 Demo 工程模板相比，简洁易懂，去掉了一些初学者不易理解或不必要的文件，同时应用底层驱动构件化的思想改进了程序结构，重新分类组织了工程，目的是引导读者进行规范的文件组织与编程。

1．工程名与新建工程

不必在意工程名，而使用工程文件夹标识工程，不同工程文件夹就区别不同工程。这样工程文件夹内的文件中所含的工程名字不再具有标识意义，可以修改，也可以不修改。建议新工程文件夹使用手动复制标准模板工程文件夹或复制功能更少的旧工程的方法来建立，这样，复用的构件已经存在。不推荐使用 KDS3.0 的新建功能来建立一个新工程。

2．工程文件夹内的基本内容

工程文件夹内编号的共含 8 个下级文件夹，除去 KDS 环境保留的文件夹 Includes 与 Debug，分别是 01_Doc、02_CPU、03_MCU、04_Linker_File、05_Driver、06_App_Component、07_Soft_Component、08_Source，其简明功能及特点见表 4-3。

表 4-3 工程文件夹内的基本内容

编号	文 件 夹	简明功能及特点
1	01_Doc	说明文档文件夹，工程改动时，及时记录
2	02_CPU	CMSIS M0+内核文件
3	03_MCU	MCU 文件夹，存放芯片头文件及芯片初始化文件，MCU 不同时，芯片头文件需更换
4	04_Linker_File	链接文件夹，放置链接文件
5	05_Driver	底层驱动文件夹，逐步加入各模块驱动构件
6	06_App_Component	存放应用构件，应用构件被定义为通过调用底层驱动构件而完成特定功能的构件，如 led、lcd、电机开关构件
7	07_Soft_Component	抽象软件构件文件夹，与硬件不直接相关的软件构件，或调用底层构件完成的功能软件构件
8	08_Source	源程序文件夹，含主程序文件、中断服务例程文件等，这些文件是工程开发人员进行编程的主要对象

4.4.2 链接文件

“..\04_Linker_File”文件夹内的 intflash.ld 文件被称为链接脚本文件，简称链接文件，可在工程环境下双击打开，默认进入的是“Configuration”界面，在该界面的最下方可切换界面，然后转入“Source”界面，即可查看源码，一般不必更改。链接文件描述了相关文件按什么顺序链接在一起，生成输出机器码文件.elf。链接文件给出了程序放在何地址、常数放在何地址、堆栈大小及地址范围等信息。表 4-4 给出了 KEA128-Light 工程的 intflash.ld 文件的简要分析，分析只抽取与程序、数据安排相关部分说明，其他部分略去，有关语法不做介绍。这个文件在同一芯片的所有工程中，原则上不改变。

MEMORY 命令告诉链接器，在整个可用空间中，哪些区域可以作为哪一类段使用。每个段名称后面的括号内字母是该段属性的标识，“R”代表该段存储单元的数据可读取（Readable），“W”代表该段存储单元可写入（Writable），“X”代表该段存储单元中放入的数据是可执行的代码（eXecutable）。

表 4-4　链接文件简明分析

顺序/命令	内　　容	简 要 说 明
1 指明程序开始点	ENTRY(Reset_Handler)	程序开始点。这里的 Reset_Handler 是指“..\03_MCU”中的 startup_SKEAZ1284.S 文件中用汇编语言编写的 Reset_Handler 函数
2 设置栈底地址	HEAP_SIZE = DEFINED(__heap_size__) ? __heap_size__ : 0x00000400; STACK_SIZE = DEFINED(__stack_size__)? __stack_size__ : 0x00000400;	若未定义堆及栈的大小，则设堆空间大小为 0x400（1 KB），设栈空间大小为 0x400（1 KB）可根据需求更改
3 MEMORY，定义和划分存储空间可用的资源	m_interrupts(rx) : ORIGIN = 0x00000000, LENGTH = 0x100	中断向量区，共 256 B，只使用到了前 192 B，48 个中断向量
	m_flash_config(rx) : ORIGIN = 0x00000400, LENGTH = 0x10	Flash 配置区
	m_text (rx) : ORIGIN = 0x00000400, LENGTH = 0x0001FBF0	Flash 用户程序区，跳过保留中断向量及 Flash 配置区后，开始存放程序、常量，长度为 0x0001FBF0
	m_data (rw) : ORIGIN = 0x1FFFF000, LENGTH = 0x00004000	RAM 数据区，长度为 16 KB。定义两个数据存储区，分别在 RAM 的 SRAM_L、SRAM_H 区域，各保留 0x410 长度的存储空间用于在上电启动后复制的中断向量表备用
4 SECTIONS，对各个块的内容的定义	__VECTOR_RAM = 0x1FFFF000; .interrupts : …….	中断向量表在 RAM 中的位置。中断向量表在 ROM 中存放在 m_interrupts 段，起始地址 0x00000000
	.cfmprotect : …….	Flash 配置域相关参数存放在 m_cfmprotrom 段中，起始地址为 0x00000400
	.text : ……	代码段。程序存放在 Flash 存储区中的.text 段。文件中的*(.text)指程序，*(.rodata)指常量
	.data: ……	标准数据段，可以用来初始化全局变量和静态变量
	.bss: ……	未初始化全局变量和静态变量段，在.data 之后

SECTIONS 命令功能强大，主要包含代码段（含常量）、标准数据段、中断向量表的存放具体位置，并定义一些符号，供其他文件使用。

通过编译输出产生的.MAP 文件可以了解代码、常量、函数、变量在存储空间中的分配情况。

链接文件中的“.”可看作一个地址变量，每当分配数据段后，“.”表示的地址将跳过分配数据段的地址区间递增，指向下一段可能分配的数据段的起始。

在 ld 文件中未指定各数据段的固定长度（除固定分配栈空间 1 KB），各数据段空间根据工程代码的实际编译结果动态分配。但观察 ld 文件中内容可知，在此分配的各数据段是连续存放的。

4.4.3　机器码文件

KDS 开发平台，针对 KEA 系列 MCU，运行 Cross ARM GCC 编译器，在编译链接过程中生成针对 ARM CPU 的.elf 格式可执行代码，同时也可生成.hex 格式的机器码。

.elf（Executable and Linking Format），即“可执行连接格式”，最初由 UNIX 系统实验室（UNIX System Laboratories，USL）作为应用程序二进制接口（Application Binary Interface，ABI）的一部分而制定和发布的。其最大特点在于它有比较广泛的适用性，通用的二进制

接口定义使之可以平滑地移植到多种不同的操作环境上。UltraEdit 软件工具查看.elf 文件内容。

.hex（Intel HEX）文件是由一行行符合 Intel HEX 文件格式的文本所构成的 ASCII 文本文件，在 Intel HEX 文件中，每一行包含一个 HEX 记录，这些记录由对应机器语言码（含常量数据）的十六进制编码数字组成。在 KDS 环境下，直接双击并单击“F5”键刷新可查看该文件。

1．记录格式

.hex 文件中的语句有六种不同类型的语句，但总体格式是一样的，根据表 4-5 格式来记录。

表 4-5 .hex 文件记录行语义

	字 段 1	字 段 2	字 段 3	字 段 4	字 段 5	字 段 6
名称	记录标记	记录长度	偏移量	记录类型	数据/信息区	校验和
长度	1 字节	1 字节	2 字节	1 字节	*N* 字节	1 字节
内容	开始标记“:”		数据类型记录有效；非数据类型，该字段为“0000”	00—数据记录；01—文件结束记录；02—扩展段地址；03—开始段地址；04—扩展线性地址；05—链接开始地址	取决于记录类型	开始标记之后字段的所有字节之和的补码。 校验和=0xFF–（记录长度+记录偏移+记录类型+数据段）+0x01

2．实例分析

以 KEA128-Light 工程中的 KEA128-Light.hex 为例，进行简明分析。截取第一个实例工程的“.hex”文件的部分行进行分解，见表 4-6。

表 4-6 KEA128_Light.hex 文件部分行分解

行	记 录 标 记	记 录 长 度	偏 移 量	记 录 类 型	数据/信息区	校 验 和
1	:	10	0000	00	00300020390C0000950C0000950C0000	19
2	:	10	0010	00	950C0000950C0000950C0000950C0000	5C
12	:	10	00B0	00	950C0000950C0000950C0000950C0000	BC
13	:	10	0800	00	80B582B000AF00F0DBF972B680239B00	A8
80	:	10	0C30	00	34F4FF1F28F4FF1F80B500AF00F00EF8	5A
177	:	00	0000	01		FF

（1）分析第 1 行，以“：”开始，长度为“0x10”（16 个字节），“0000”表示相对地址，紧接着的“00”代表记录类型为数据类型，接下来的就是数据段“00300020390C0000950C0000950C0000”，该数据段存放在偏移地址为“0000”的存储区的机器操作码，也就是说，只有这些数据被写入到 Flash 存储区。值得注意的是，这里的 hex 文件中，数据部分是以小端方式存储的，这与 MCU 内部的存储方式有关。在这种格式中，字的低字节存储在低地址中，而字的高字节存放在高地址中，第 1 个 4 字节“00 30 00 20”，实际表示的数据内容为“20 00 30 00”，就是堆栈栈顶（=RAM 最高地址+1），参见链接文件与映像文件，这 4 个字节也就是中断向量表中开始内容（占用了 0 号中断位置），其内容在 MCU 启动时被放入堆栈寄存器 SP 中。接下来的 4 个字节“39 0C 00 00”→“00 00 0C39”（参见第 80 行），占用中断向量表 1 号中断位

置，其地址减 1 的内容在 MCU 启动时被放入程序堆栈计数器 PC 中，也就是开始执行代码的存放处，即启动函数__thumb_startup()在 Flash 中存放的开始地址（可以从“.map”文件、“.lst”文件找到相应信息进行理解，例如，此时__thumb_startup()的地址为 0x00000C38）。原因在于 Cortex-M0+处理器的指令地址为半字对齐，也就意味着 PC 寄存器的最低位必须始终为 0。但是，程序在跳转时，PC 的最低位必须被置为 1，以表明内核仍然处于 Thumb 状态。而不是 ARM 状态，参见《ARM-Cortex-M0 权威指南》。

（2）从第 1 行后半部开始至 12 行，是中断向量表，从第 13 行开始是程序记录。

（3）第 220 行（最后一行）为文档的结束记录，记录类型为“0x01”；“0xFF”为本记录的校验和字段内容。

综合分析工程的.map 文件、.ld 文件、.hex 文件、.lst 文件，可以理解程序的执行过程，也可以对生成的机器码进行分析对比。

4.4.4　其他相关文件功能简介

1．中断服务例程文件（isr.h、isr.c）

位于“..\08_Sources”文件夹下的 isr.h、isr.c 文件是中断处理相关文件，有关中断机制、中断文件组织方式及编程方法将在第 6 章阐述。

2．启动文件（startup_SKEAZ1284.S）

启动文件 startup_SKEAZ1284.S，位于“..\03_MCU”文件夹下，主要提供 main()函数之前的代码 Reset_Handler，该函数内容分为：

（1）关闭所有外设中断并清除所有外设挂起中断标志；

（2）关看门狗，配置时钟；

（3）将 ROM 中的初始化数据拷贝到 RAM 中；

（4）将未初始化数据 BSS 段清 0。

（5）转至 main()继续执行。

除此之外，启动文件 startup_SKEAZ1284.S 中还存放了芯片的中断向量表，并用弱符号方式为内核异常及外设中断定义了默认的中断处理函数。

3．系统初始化文件（system_SKEAZ1284.h、system_SKEAZ1284.c）

系统初始化文件 system_SKEAZ1284.h、system_SKEAZ1284.c 位于“..\03_MCU”文件夹下，主要提供启动文件中调用的系统初始化函数 SystemInit()及其相关宏常量的定义，此函数实现关闭看门狗及配置系统时钟的功能。

4．总头文件与主程序文件（includes.h、main.c）

总头文件 includes.h 是 main.c 使用的头文件，内含外部函数引用、常量、全局变量声明和外部变量引用。main.c 文件是工程任务的核心文件，main 函数在该文件中实现。在 main

函数中包含了一个永久循环，对具体事务过程的操作几乎都是添加在该主循环中，含有操作系统下编程的调度作用。

5．内核及芯片相关文件（core_cm0plus.h、core_cmFunc.h、core_cmInstr.h、SKEAZ1284.h）

内核相关文件 core_cm0plus.h、core_cmFunc.h 及 core_cmInstr.h 位于“..\02_CPU”文件夹下，它们是 ARM 公司提供的符合 CMSIS（Cortex Microcontroller Software Interface Standard，ARM Cortex，微控制器软件接口标准）的内核相关头文件，与供应商无关。其中 core_cm0plus.h 为 ARM Cortex-M0+内核的核内外设访问层头文件，而 core_cmFunc.h 及 core_cmInstr.h 则分别为 ARM Cortex-M 系列内核函数及指令访问头文件。使用 CMSIS 标准可简化程序的开发流程，提高程序的可移植性。

芯片头文件 SKEAZ1284.h 位于“..\03_MCU”文件夹下，在 SKEAZ1284.h 文件中，给出了 KEA128 芯片专用的寄存器地址映射，由芯片厂家提供。需要注意的是，针对某一款芯片进行开发时，有专用的寄存器映射文件，由芯片设计人员提供，一般嵌入式应用开发者不必修改该文件。

6．.map 文件、.lst 文件

.map 文件与.lst 文件位于“..\Debug”文件夹，由编译链接产生。.map 文件提供了查看程序、堆栈设置、全局变量、常量等存放的地址信息。.map 文件中指定的地址在一定程度上是动态分配的（由编译器决定），工程有任何修改，这些地址都可能发生变动。.lst 文件提供了函数编译后，机器码与源代码的对应关系，用于程序分析。

4.4.5 芯片上电启动执行过程

1．中断向量表存放位置的分析理解

先查看工程文件夹中“..\04_Linker_File\intflash.ld”文件中 MEMORY 部分，有下列语句。

```
m_interrupts (RX) : ORIGIN = 0x00000000, LENGTH = 0x00000100
```

该语句确定了 m_interrupts 代表从 0x00000000 开始，长度为 0x00000100。在随后的 SECTIONS 部分有下列语句：

```
.interrupts :
{
    __VECTOR_TABLE = .;
    . = ALIGN(4);
    KEEP(*(.isr_vector))     /*Startup code*/
    . = ALIGN(4);
} > m_interrupts
```

这样*(.isr_vector)指向 0x00000000 地址。

再查看工程文件夹中“..\ 03_MCU\startup_SKEAZ1284.S”文件，该文件中关于中断向量表的内容如下：

```
    ……
.section .isr_vector, "a"
    .align 2
    .globl __isr_vector
__isr_vector:                           //中断向量表
    .long   __StackTop                  /*Top of Stack*/
    .long   Reset_Handler               /*Reset Handler*/
    .long   NMI_Handler                 /*NMI Handler*/
    .long   HardFault_Handler           /*Hard Fault Handler*/
    .long   0                           /*Reserved*/
    .long   0                           /*Reserved*/
    .long   0                           /*Reserved*/
    .long                               /*Reserved*/
    .long   0                           /*Reserved*/
    .long   0                           /*Reserved*/
    .long   0                           /*Reserved*/
    .long   SVC_Handler                 /*SVCall Handler*/
    .long   0                           /*Reserved*/
    .long   0                           /*Reserved*/
    .long   PendSV_Handler              /*PendSV Handler*/
    .long   SysTick_Handler             /*SysTick Handler*/

                                        /*External Interrupts*/
    .long   Reserved16_IRQHandler       /*Reserved interrupt*/
    .long   Reserved17_IRQHandler       /*Reserved interrupt*/
    .long   Reserved18_IRQHandler       /*Reserved interrupt*/
    .long   Reserved19_IRQHandler       /*Reserved interrupt*/
    .long   Reserved20_IRQHandler       /*Reserved interrupt*/
    .long   FTMRE_IRQHandler    /*Command complete and read collision*/
    .long   PMC_IRQHandler      /*Low-voltage detect, low-voltage warning*/
    .long   IRQ_IRQHandler              /*External Interrupt*/
    .long   I2C0_IRQHandler             /*I2C0 interrupt*/
    .long   I2C1_IRQHandler             /*I2C1 interrupt*/
    .long   SPI0_IRQHandler     /*SPI0 single interrupt vector for all sources*/
    .long   SPI1_IRQHandler     /*SPI1 single interrupt vector for all sources*/
    .long   UART0_IRQHandler            /*UART0 status and error*/
    .long   UART1_IRQHandler            /*UART1 status and error*/
    .long   UART2_IRQHandler            /*UART2 status and error*/
    .long   ADC_IRQHandler              /*ADC interrupt*/
    .long   ACMP0_IRQHandler            /*ACMP0 interrupt*/
    .long   FTM0_IRQHandler     /*FTM0 single interrupt vector for all sources*/
    .long   FTM1_IRQHandler     /*FTM1 single interrupt vector for all sources*/
    .long   FTM2_IRQHandler     /*FTM2 single interrupt vector for all sources*/
    .long   RTC_IRQHandler              /*RTC overflow*/
    .long   ACMP1_IRQHandler            /*ACMP1 interrupt*/
    .long   PIT_CH0_IRQHandler          /*PIT CH0 overflow*/
    .long   PIT_CH1_IRQHandler          /*PIT CH1 overflow*/
    .long   KBI0_IRQHandler             /*Keyboard interrupt0(32bit)*/
    .long   KBI1_IRQHandler             /*Keyboard interrupt1(32bit)*/
```

```
    .long   Reserved42_IRQHandler       /*Reserved interrupt*/
    .long   ICS_IRQHandler              /*ICS interrupt*/
    .long   WDOG_IRQHandler             /*Watchdog timeout*/
    .long   PWT_IRQHandler              /*PWT interrupt*/
    .long   MSCAN_RX_IRQHandler         /*MSCAN Rx Interrupt*/
    .long   MSCAN_TX_IRQHandler         /*MSCAN Tx, Err and Wake-up interrupt*/

.size    __isr_vector, . - __isr_vector
……
.align1
.thumb_func
.weakDefault_Handler
.typeDefault_Handler, %function
Default_Handler:
b.
.sizeDefault_Handler, . - Default_Handler

/*Macro to define default handlers. Default handler
 *will be weak symbol and just dead loops. They can be
 *overwritten by other handlers*/
.macrodef_irq_handlerhandler_name
.weak\handler_name
.set\handler_name, Default_Handler
.endm

/*Exception Handlers*/
   def_irq_handler   NMI_Handler
   def_irq_handler   HardFault_Handler
   def_irq_handler   SVC_Handler
   def_irq_handler   PendSV_Handler
   def_irq_handler   SysTick_Handler
   def_irq_handler   Reserved16_IRQHandler
   def_irq_handler   Reserved17_IRQHandler
   def_irq_handler   Reserved18_IRQHandler
   def_irq_handler   Reserved19_IRQHandler
   def_irq_handler   Reserved20_IRQHandler
   def_irq_handler   FTMRE_IRQHandler
   def_irq_handler   PMC_IRQHandler
   def_irq_handler   IRQ_IRQHandler
   def_irq_handler   I2C0_IRQHandler
   def_irq_handler   I2C1_IRQHandler
   def_irq_handler   SPI0_IRQHandler
   def_irq_handler   SPI1_IRQHandler
   def_irq_handler   UART0_IRQHandler
   def_irq_handler   UART1_IRQHandler
   def_irq_handler   UART2_IRQHandler
   def_irq_handler   ADC_IRQHandler
   def_irq_handler   ACMP0_IRQHandler
   def_irq_handler   FTM0_IRQHandler
```

```
        def_irq_handler    FTM1_IRQHandler
        def_irq_handler    FTM2_IRQHandler
        def_irq_handler    RTC_IRQHandler
        def_irq_handler    ACMP1_IRQHandler
        def_irq_handler    PIT_CH0_IRQHandler
        def_irq_handler    PIT_CH1_IRQHandler
        def_irq_handler    KBI0_IRQHandler
        def_irq_handler    KBI1_IRQHandler
        def_irq_handler    Reserved42_IRQHandler
        def_irq_handler    ICS_IRQHandler
        def_irq_handler    WDOG_IRQHandler
        def_irq_handler    PWT_IRQHandler
        def_irq_handler    MSCAN_RX_IRQHandler
        def_irq_handler    MSCAN_TX_IRQHandler
        def_irq_handler    DefaultISR

        .end
```

2．中断向量表内容的理解

中断向量表第 1 行“__StackTop”为栈顶指针，存放在 0x00000000～0x00000003 的地址内，其值不固定。第 2 行“Reset_Handler”即地址 0x00000004～0x00000007，地址内容为启动函数 Reset_Handler 的首地址。随后地址依次存放各中断例程的起始地址，关于中断有关内容见第 6 章。

3．main 函数之前的程序运行过程分析

芯片复位到 main 函数之前程序运行过程如下。

（1）芯片上电复位后，芯片内部机制首先从 Flash 的 0x00000000 地址中，取出第一个表项的内容，赋给内核寄存器 SP（堆栈指针），完成堆栈指针初始化。例如，在链接文件 intflash.ld 中，链接时将__StackTop 的值链接到 Flash 的 0x00000000 地址中。

（2）芯片内部机制将第二个表项的内容，赋给内核寄存器 PC（程序计数器）。

（3）由于该表项存放启动函数 Reset_Handler 的首地址，因而转到工程文件夹中“..\03_MCU\startup.c“文件中，执行函数 startup。

（4）在函数 startup_SKEAZ1284.S 中，依次执行关闭所有外设中断并清除所有外设挂起中断标志、关闭看门狗、系统时钟初始化[①]、将 ROM 中的初始化数据拷贝到 RAM 中、清零未初始化 BSS 数据段、进入主函数 main。

在实际应用中，可根据是否启动看门狗、是否复制中断向量表至 RAM、是否清零未初始化 BSS 数据段等要求，修改此文件。在未理解相关内容情况下，不建议修改 startup_SKEAZ1284.S 及 system_SKEAZ1284.c 文件。主函数 main 中的永久循环之前的代码顺序，有一定规范，将在第 6 章第一个中断程序中阐述。

① 系统时钟详见 13.1 节。本书的所用例程均使用片内时钟源，经系统时钟初始化程序后，内核时钟为 48 MHz，总线时钟为 24 MHz。

4.5 第一个汇编语言工程：控制小灯闪烁

汇编语言编程给人的第一种感觉就是难，相对于 C 语言编程，汇编语言在编程的直观性、编程效率及可读性等方面都有所欠缺，但掌握基本的汇编语言编程方法是嵌入式学习的基本功，可以增加嵌入式编程者的“内力”。

在本书教学资料中提供的 KDS 开发环境中，汇编程序是通过新建工程的方式并且经过修改芯片初始化组织起来的。汇编工程通常包含芯片相关的程序框架文件、软件构件文件、工程设置文件、主程序文件及抽象构件文件等。下面将结合第一个 KEA128 汇编工程实例“KEA128_Light_ASM”，讲解上述的文件概念，并详细分析 KEA128 汇编工程的组成、汇编程序文件的编写规范、软硬件模块的合理划分等。读者若能认真分析与实践第一个汇编实例程序，可以达到由此入门的目的。

4.5.1 汇编工程文件的组织

汇编工程的样例在“..\KEA128_Light_ASM”文件夹中。本汇编工程类似 C 工程，仍然按构件方式进行组织。图 4-2 给出了小灯闪烁汇编工程的树形结构，主要包括 MCU 相关头文件夹、底层驱动构件文件夹、Debug 工程输出文件夹、程序文件夹等。读者按照理解 C 工程的方式，理解这个结构。

KEA128_Light_ASM	工程名
Binaries	
Includes	
01_Doc	文档文件夹
02_CPU	内核头文件
03_MCU	MCU 相关文件
SKEAZ1284.h	KEAZ128 芯片头文件
startup_SKEAZ1284.S	启动代码
system_SKEAZ1284.c	系统初始化源文件
system_SKEAZ1284.h	系统初始化头文件
04_Linker_File	链接文件夹
intflash.ld	链接文件
05_Driver	驱动文件夹
gpio	GPIO 底层构件文件夹
gpio.S	GPIO 底层构件
gpio.inc	
06_App_Component	应用构件文件夹
light	小灯构件文件夹
light.S	小灯构件
light.inc	软件构件文件夹
07_Soft_Component	通用代码文件夹
common	源文件夹
08_Sources	总头文件
include.S	主函数
main.S	工程输出
Debug	总头文件

图 4-2 小灯闪烁汇编工程的树形结构

汇编工程仅包含一个汇编主程序文件，该文件名固定为 main.s。汇编程序的主体是程序的主干，要尽可能简洁、清晰、明了，程序中的其余功能，尽量由子程序去完成，主程序主要完成对子程序的循环调用。主程序文件 main.s 包含以下内容。

部分文件包含内容描述如下。

（1）01_Doc：工程名、程序描述、版本、日期等。若调试过程有新的体会，也可在此添加，目的是为将来自己使用，或为同组开发提供必要的备忘信息。

（2）总头文件：声明全局变量和包含主程序文件中需要的头文件、宏定义等。

（3）主函数：主程序一般包括初始化与主循环两大部分。初始化包括堆栈初始化、系统初始化、I/O 端口初始化、中断初始化等；主循环是程序的工作循环，根据实际需要安排程序段，但一般不宜过长，建议不要超过 100 行，具体功能可通过调用子程序来实现，或由中断程序实现。

4.5.2　Light 构件汇编程序 light.s

Light 构件用于控制指示灯的亮或暗，包括小灯初始化程序 light_init、控制小灯亮暗程序 light_control，以及切换小灯亮暗程序 light_change。

```
#=============================================================================
#文件名称：light.s
#功能概要：小灯驱动程序文件
#=============================================================================
#include "light.inc"

#=============================================================================
# 函数名称：light_init
# 函数返回：无
# 参数说明：r0:(端口号)|(引脚号)（例：PORTB|(5) 表示为 B 口 5 号脚）
#          r3:设定小灯状态。由 light.inc 中宏定义。
# 功能概要：指示灯驱动初始化
#=============================================================================
light_init:
    push {r0-r3,lr}              @保存现场，将下一条指令地址入栈
    mov r2,#1                    @小灯为输出
    bl gpio_init                 @调用 gpio 初始化函数
    pop {r0-r3,pc}               @恢复现场，返回主程序处继续执行

#=============================================================================
# 函数名称：light_control
# 函数返回：无
# 参数说明：r0：(端口号)|(引脚号)（例：PORTB|(5) 表示为 B 口 5 号脚）
#          r3：设定小灯状态。由 light.inc 中宏定义。
# 功能概要：控制指示灯亮暗。
#=============================================================================
light_control:
    push {r0-r3,lr}
```

```
    mov r2,#1                    @小灯为输出
    bl gpio_set                  @调用 gpio 引脚设置函数
    pop {r0-r3,pc}

#=========================================================================
# 函数名称：light_change
# 函数返回：无
# 参数说明：r0:(端口号)|(引脚号)（例：PORTB|(5) 表示为 B 口 5 号脚）
# 功能概要：切换指示灯亮暗。
#=========================================================================
light_change:
    push {r0-r3,lr}
    bl gpio_reverse              @调用后 gpio 引脚反转函数
    pop {r0-r3,pc}
```

4.5.3 Light 测试工程主程序及汇编工程执行过程

在 include.s 中，包含各个构件的汇编文件，本工程中包含 light.s，这样在工程中就可以调用 light 构件的接口函数。首先调用 light_Init 函数，初始化所需的指示灯。注意初始化时，要让每一盏灯初始状态为“暗”。随后，通过 light_control 函数控制指示灯亮、暗。通过变量的递增并且设置频率后，就能够在程序运行中看到指示灯进行闪烁的现象。

```
#=========================================================================
#文件名称：main.s
#功能概要：汇编编程控制小灯闪烁
#版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
#版本更新：2015-04-25
#=========================================================================
#include "include.S"

#start 主函数定义开始
    .section   .text.main
    .global main                 @定义全局变量，在芯片初始化之后调用
    .align 2                     @指令对齐
    .type main function          @定义主函数类
    .align 2
#end  主函数定义结束
main:
    cpsid i                      @关闭总中断
    #小灯初始化, r0,r3 是 light_init 的入口参数
    ldr r0,=RUN_LIGHT1           @r0 指明端口和引脚（用=是因为宏常数>=256,且用 ldr)
    mov r3,#LIGHT_OFF            @r3 指明引脚的初始状态
    bl light_init                @调用小灯初始化函数
cpsie  i                         @开总中断
#主循环开始=============================================================
main_loop1:
    ldr r4,=RUN_COUNTER_MAX      @取延时值到 r4
    mov r5,#0                    @从零计数
```

```
loop:
    add r5,#1                    @加 1 计数
    cmp r4,r5                    @r4 值与 r5 值比较
    bne loop                     @不相等，则跳转 loop
    bl light_change              @相等，则调用小灯亮暗转变函数
    bne main_loop1               @跳转 main_loop1
#主循环结束=================================================================
.end
```

当 KEA128 芯片内电复位或热复位后，系统程序的执行流程如下：从复位向量处取出程序执行的首地址，跳转并按该地址执行；执行 startup.c 文件中的__thumb_startup 函数，进行系统的初始化，在系统带电的状态下，硬件中断机制始终开启，并实时地“监听”内外环境而恰当地激发特定的事务处理过程。具体过程如下。

首先系统在加电过后，芯片内的硬件机制会产生加电复位中断，这时系统到向量表中查找复位向量地址，并转向这个地址继续执行。在本书所有工程样例，到 vectors.c 文件中都可以找到中断向量表，在该表的第一行是复位中断向量地址：VECTOR 0 __thumb_startup（复位向量）。__thumb_startup()函数执行的操作是关开门狗、bss 区清零、时钟初始化等。其次，执行 main.s 文件中的 main 函数，对所用到的模块进行初始化、使能中断等。最后当某个中断发生后，MCU 将转到中断向量表文件 isr.asm 所指定的中断入口地址处开始执行中断服务程序（Interrupt Service Routine，ISR），本书不再描述汇编中断程序，深入学习的读者，不难完成此任务。

第 5 章

嵌入式硬件构件与底层驱动构件基本规范

本章导读：本章主要分析嵌入式系统构件化设计的重要性和必要性，给出嵌入式硬件构件和底层驱动构件的设计方法和原则，详细阐述构件的编程思想和编程规范，以及构件的重用与移植方法。主要内容有：分析了嵌入式构件化必要性，给出了构件化的定义；给出了基于嵌入式硬件构件的电路原理图设计简明规则及使用硬件构件组装系统的方法；给出了嵌入式底层驱动构件的概念及层次模型；给出了构件设计的基本思想与基本原则，给出了编码风格、构件头文件及源程序文件的设计规范；给出了硬件构件及底层软件构件的重用与移植方法。这一章的目的就是期望通过一定的规范，提高嵌入式软硬件设计的可重用性和可移植性。

5.1 嵌入式硬件构件

机械、建筑等传统产业的运作模式是先生产符合标准的构件（零部件），然后将标准构件按照规则组装成实际产品。其中，构件（Component）是核心和基础，复用是必需的手段。传统产业的成功充分证明了这种模式的可行性和正确性。软件产业的发展借鉴了这种模式，为标准软件构件的生产和复用确立了举足轻重的地位。

随着微控制器及应用处理器内部 Flash 存储器可靠性提高及擦写方式的变化，内部 RAM 及 Flash 存储器容量的增大，以及外部模块内置化程度的提高，嵌入式系统的设计复杂性、设计规模及开发手段已经发生了根本变化。在嵌入式系统发展的最初阶段，嵌入式系统硬件和软件设计通常是由一个工程师来承担的，软件在整个工作中的比例很小。随着时间的推移，硬件设计变得越来越复杂，软件的份量也急剧增长，嵌入式开发人员也由一人发展为由若干人组成的开发团队。为此希望提高软硬件设计可重用性与可移植性，构件的制作与应用是重用与移植的基础与保障。

5.1.1 嵌入式硬件构件的概念

要提高硬件设计可重用性与可移植性，就必须有工程师们共同遵守的硬件设计规范。设计人员若凭借个人工作经验和习惯的积累进行系统硬件电路的设计，在开发完一个嵌入式应用系统再进行下一个应用开发时，硬件电路原理图往往需要从零开始，重新绘制；或者在一个类似的原理图上修改，但往往又很麻烦，容易出错。因此把构件的思想引入到硬件原理图设计中。

什么是嵌入式硬件构件？它与我们常说的硬件模块有什么不同？

众所周知，嵌入式硬件是任何嵌入式产品不可分割的重要组成部分，是整个嵌入式系统

的构建基础，嵌入式应用程序和操作系统都运行在特定的硬件体系上。一个以 MCU 为核心的嵌入式系统通常包括电源、写入器接口电路、硬件支撑电路、UART、USB、Flash、A/D、D/A、LCD、键盘、传感器输入电路、通信电路、信号放大电路、驱动电路等硬件模块，其中有些模块集成在 MCU 内部，有的位于 MCU 之外。

与硬件模块的概念不同，**嵌入式硬件构件是指将一个或多个硬件功能模块、支撑电路及其功能描述封装成一个可重用的硬件实体，并提供一系列规范的输入/输出接口**。由定义可知，传统概念中的硬件模块是硬件构件的组成部分，一个硬件构件可能包含一个硬件功能模块，也有可能包含多个。

根据接口之间的生产消费关系，**接口可分为提供接口和需求接口两类**。根据所拥有接口类型的不同，硬件构件分为核心构件、中间构件和终端构件三种类型。**核心构件**只有提供接口，没有需求接口，也就是说，它只为其他硬件构件提供服务，而不接受服务。在以单 MCU 为核心的嵌入式系统中，MCU 的最小系统就是典型的核心构件。**中间构件**既有需求接口又有提供接口，即它不仅能够接受其他构件提供的服务，而且也能够为其他构件提供服务。而**终端构件**只有需求接口，它只接受其他构件提供的服务。这三种类型构件的区别如表 5-1 所示。

表 5-1 核心构件、中间构件和终端构件的区别

类 型	需 求 接 口	提 供 接 口	举 例
核心构件	无	有	芯片的硬件最小系统
中间构件	有	有	电源控制构件、232 电平转换构件
终端构件	有	无	LCD 构件、LED 构件、键盘构件

利用硬件构件进行嵌入式系统硬件设计之前，应该进行硬件构件的合理划分，按照一定规则，设计与系统目标功能无关的构件个体，然后进行“组装”，完成具体系统的硬件设计。这样，这些构件个体也可以被组装到其他嵌入式系统中。在硬件构件被应用到具体系统中后，设计人员需要做的仅仅是为需求接口添加接口网标。

5.1.2 基于嵌入式硬件构件的电路原理图设计简明规则

1．硬件构件设计的通用规则

在设计硬件构件的电路原理图时，需遵循以下基本原则。

（1）元器件命名格式：对于核心构件，其元器件直接编号命名，同种类型的元件命名时冠以相同的字母前缀。例如，电阻名称为 R1、R2 等，电容名称为 C1、C2 等，电感名称为 L1、L2 等，指示灯名称为 E1、E2 等，二极管名称为 D1、D2 等，三极管名称为 Q1、Q2 等，开关名称为 K1、K2 等。对于中间构件和终端构件，其元器件命名格式采用“构件名-标志字符?”。例如，LCD 构件中所有的电阻名称统一为“LCD-R?”，电容名称统一为“LCD-C?”。当构件原理图应用到具体系统中时，可借助原理图编辑软件为其自动编号。

（2）为硬件构件添加详细的文字描述，包括中文名称、英文名称、功能描述、接口描述、注意事项等，以增强原理图的可读性。中英文名称应简洁明了。

（3）将前两步产生的内容封装在一个虚线框内，组成硬件构件的内部实体。

（4）为该硬件构件添加与其他构件交互的输入/输出接口标识。接口标识有两种：接口注释和接口网标。它们的区别是：接口注释标于虚线框以内，是为构件接口所作的解释性文字，目的是帮助设计人员在使用该构件时理解该接口的含义和功能；而接口网标位于虚线框之外，且具有电气特性。为使原理图阅读者便于区分，接口注释采用斜体字。

在进行核心构件、中间构件和终端构件的设计时，除了要遵循上述的通用规则外，还要兼顾各自的接口特性、地位和作用。

2．核心构件设计规则

设计核心构件时，需考虑的问题是：“核心构件能为其他构件提供哪些信号？”。核心构件其实就是某型号 MCU 的最小系统。**核心构件设计的目标是：凡是使用该 MCU 进行硬件系统设计时，核心构件可以直接“组装”到系统中，无须任何改动**。为了实现这一目标，在设计核心构件的实体时必须考虑细致、周全，包括稳定性、扩展性等，封装要完整。核心构件的接口都是为其他构件提供服务的，因此接口标识均为接口网标。在进行接口设计时，需将所有可能使用到的引脚都标注上接口网标（不要考虑：核心构件将会用到怎样的系统中去）。若同一引脚具有不同功能，则接口网标依据第一功能选项命名。遵循上述规则设计核心构件的好处是：当使用核心构件和其他构件一起组装系统时，只要考虑其他构件将要连接到核心构件的哪个接口（不是考虑：核心构件将要连接到其他构件的哪个接口），这也符合设计人员的思维习惯。

3．中间构件设计规则

设计中间构件时，需考虑的问题是：“中间构件需要接收哪些信号，以及提供哪些信号？”。中间构件是核心构件与终端构件之间通信的桥梁。在进行中间构件的实体封装时，实体的涉及范围应从构件功能和编程接口两方面考虑。一个中间构件应具有明确的且相对独立的功能，它既要有接受其他构件提供的服务的接口，即需求接口，又要有为其他构件提供服务的接口，即提供接口。**描述需求接口采用接口注释，描述提供接口采用接口网标**。当中间构件作为一个“零件”组装到具体系统中时，设计人员只要考虑为构件提供服务的来源，为接口注释添加对应的应用网标即可，其他内容无须关心或改动。

中间构件的接口数目没有核心构件那样丰富。为直观起见，设计中间构件时，将构件的需求接口放置在构件实体的左侧，提供接口放置在右侧。接口网标的命名规则是：构件名称-引脚信号/功能名称。而接口注释名称前的构件名称可有可无，它的命名隐含了相应的引脚功能。

电源控制构件（如图 5-1 所示）、可变频率产生构件（如图 5-2 所示）是常用的中间构件。图 5-1 中的 *Power-IN* 和图 5-2 中的 *SDI*、*SCK* 和 *SEN* 均为接口注释，Power-OUT 和 LTC6903-OUT 为接口网标。

4．终端构件设计规则

设计终端构件时，需考虑的问题是：“终端构件需要什么信号才能工作？”。终端构件是嵌入式系统中最常见的构件，它没有提供接口，仅有与上一级构件交互的需求接口，因而接口标识均为斜体标注的接口注释。LCD（YM1602C）构件（如图 5-3 所示）、LED 构件、指示灯构件及键盘构件（如图 5-4）等都是典型的终端构件。

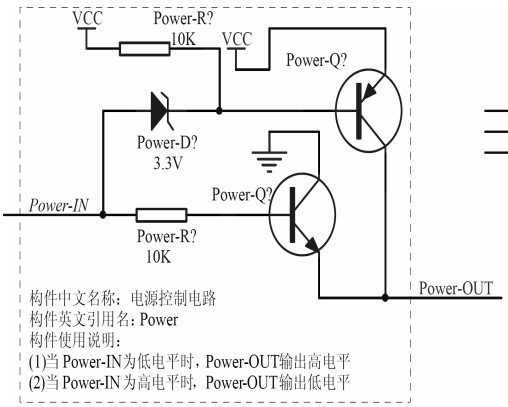


图 5-1 电源控制构件

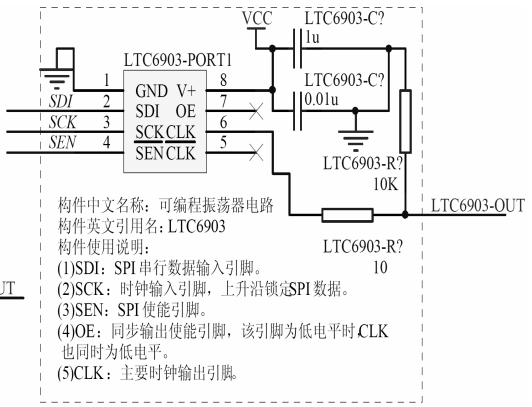


图 5-2 可变频率产生构件

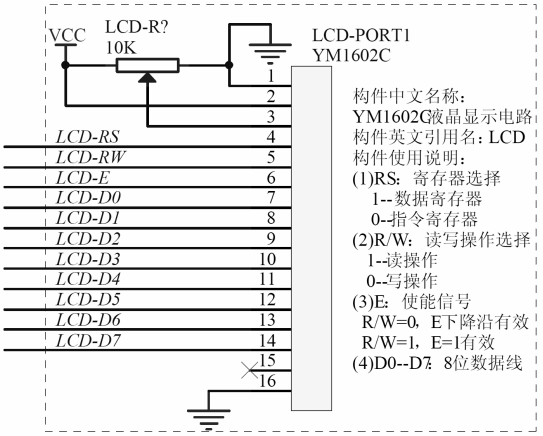


图 5-3 LCD 构件



图 5-4 键盘构件

5．使用硬件构件组装系统的方法

对于核心构件，在应用到具体的系统中时，不必作任何改动。具有相同 MCU 的应用系统，其核心构件完全相同。对于中间构件和终端构件，在应用到具体的系统中时，仅需为需求接口添加接口网标，在不同的系统中，接口网标名称不同，但构件实体内部完全相同。

使用硬件构件化思想设计嵌入式硬件系统的过程与步骤是：

（1）根据系统的功能划分出若干个硬件构件。

（2）将所有硬件构件原理图“组装”在一起。

（3）为中间构件和终端构件添加接口网标。

5.2 嵌入式底层驱动构件的概念与层次模型

嵌入式系统是软件与硬件的综合体，硬件设计和软件设计相辅相成。嵌入式系统中的驱动程序是直接工作在各种硬件设备上的软件，是硬件和高层软件之间的桥梁。正是通过驱动程序，各种硬件设备才能正常运行，达到既定的工作效果。

5.2.1　嵌入式底层驱动构件的概念

要提高软件计可重用性与可移植性，就必须充分理解和应用软件构件技术。“提高代码质量和生产力的唯一最佳方法就是**复用**好的代码”，软件构件技术是软件复用实现的重要方法，也是软件复用技术研究的重点。构件（Component）是可重用的实体，它包含了合乎规范的接口和功能实现，能够被独立部署和被第三方组装[①]。软件构件（Software Component）是指，在软件系统中具有相对独立功能、可以明确辨识构件实体。在嵌入式系统领域中，嵌入式软件构件（Embedded Software Component）是实现一定嵌入式系统功能的一组封装的、规范的、可重用的、具有嵌入特性的软件构件单元，是组织嵌入式系统功能的基本单位。嵌入式软件分为高层软件构件和底层软件构件（底层驱动构件）。高层软件构件与硬件无关，例如，实现嵌入式软件算法的算法构件、执行队列元素操作的队列构件等；而底层驱动构件与硬件密不可分，是硬件驱动程序的构件化封装。

5.2.2　嵌入式硬件构件和软件构件的层次模型

嵌入式软件构件（Embedded Software Component，ESC）是实现一定嵌入式系统功能的一组封装的、规范的、可重用的、具有嵌入特性的软件单元，是组织嵌入式系统的功能单位。

嵌入式软件构件分为高层软件构件和底层软件构件（以下简称高层构件和底层构件）。高层构件与硬件无关，而底层构件与硬件密不可分，是硬件驱动程序的封装。前面提到，在硬件构件中，核心构件为 MCU 的最小系统。通常，MCU 内部包含有 GPIO（即通用 IO）口和一些内置功能模块，可将通用 I/O 口的驱动程序封装为 GPIO 驱动构件，各内置功能模块的驱动程序封装为功能构件，如芯片内含模块的功能构件有串行通信构件、Flash 构件、定时器构件等。

在硬件构件层中，相对于核心构件而言，中间构件和终端构件是核心构件的“外设”。由这些“外设”的驱动程序封装而成的软件构件称为底层外设构件。注意，并不是所有的中间构件和终端构件都可以作为编程对象。例如，键盘、LED、LCD 等硬件构件与编程有关，而电平转换硬件构件就与编程无关，因而不存在相应的底层驱动程序，当然也就没有相应的软件构件。嵌入式硬件构件与软件构件的层次模型如图 5-5 所示。

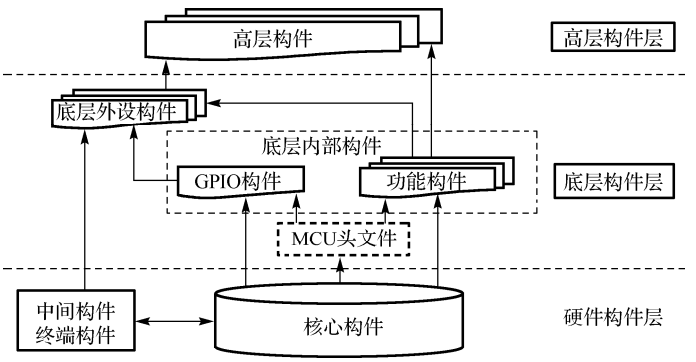


图 5-5　嵌入式硬件构件与软件构件的层次模型

① NATO Communications and Information Systems Agency. NATO Standard for Development of Reusable Software Components[S], 1991.

由图 5-5 可看出，底层外设构件可以调用底层内部构件，如 LCD 构件可以调用 GPIO 驱动构件、PCF8563 构件（时钟构件）可以调用 I2C 构件等。而高层构件可以调用底层外设构件和底层内部构件中的功能构件，而不能直接调用 GPIO 驱动构件。另外，考虑到几乎所有的底层内部构件都涉及 MCU 各种寄存器的使用，因此将 MCU 的所有寄存器定义组织在一起，形成 MCU 头文件，以便其他构件头文件中包含该头文件。



5.3 底层驱动构件的封装规范

驱动程序的开发在嵌入式系统的开发中具有举足轻重的地位，驱动程序的好坏直接关系着整个嵌入式系统的稳定性和可靠性。然而，开发出完备、稳定的驱动程序并非易事。为了提高底层驱动构件的可移植性和可复用性，特制定本规范。

5.3.1 构件设计的基本思想与基本原则

1．构件设计的基本思想

底层构件是与硬件直接打交道的软件，它被组织成具有一定独立性的功能模块，由头文件（.h）和源程序文件（.c）两部分组成。构件的头文件名和源程序文件名一致，且为构件名。

构件的头文件中主要包含必要的引用文件、描述构件功能特性的宏定义语句，以及声明对外接口函数。良好的构件头文件应该成为构件使用说明，不需要使用者查看源程序。

构件的源程序文件中包含构件的头文件、内部函数的声明、对外接口函数的实现。

将构件分为头文件与源程序文件两个独立的部分，意义在于，头文件中包含对构件的使用信息的完整描述，为用户使用构件提供充分必要的说明，构件提供服务的实现细节被封装在源程序文件中；调用者通过构件对外接口获取服务，而不必关心服务函数的具体实现细节。这就是构件设计的基本内容。

在制作底层构件时，最关键的工作是要对构件的共性和个性进行分析，设计出合理、必要的对外接口函数及其形参。尽量做到：当一个底层构件应用到不同系统中时，仅需修改构件的头文件，对于构件的源程序文件则不必修改或改动很小。

2．构件设计的基本原则

在嵌入式软件领域中，由于软件与硬件紧密联系的特性，使得与硬件紧密相连的底层驱动构件的生产成为嵌入式软件开发的重要内容之一，良好的底层驱动构件具备如下特性。

（1）**封装性**。在内部封装实现细节，采用独立的内部结构以减少对外部环境的依赖。调用者只通过构件接口获得相应功能，内部实现的调整将不会影响构件调用者的使用。

（2）**描述性**。构件必须提供规范的函数名称、清晰的接口信息、参数含义与范围、必要的注意事项等描述，为调用者提供统一、规范的使用信息。

（3）**可移植性**。底层构件的可移植性是指同样功能的构件，如何做到不改动或少改动，而方便地移植到同系列及不同系列芯片内，减少重复工作。

（4）**可复用性**。在满足一定使用要求时，构件不经过任何修改就可以直接使用。特别是

使用同一芯片开发不同项目，底层驱动构件应该做到复用。可复用性使得高层调用者对构件的使用不因底层实现的变化而有所改变。可复用性提高了嵌入式软件的开发效率、可靠性与可维护性，不同芯片的底层驱动构件复用需在可移植性基础上进行。

为了使构件设计满足封装性、描述性、可移植性、可复用性的基本要求，嵌入式底层驱动构件的开发，应遵循层次化、易用性、鲁棒性及对内存的可靠使用原则。

1）层次化原则

层次化设计要求清晰地组织构件之间的关联关系。底层驱动构件与底层硬件打交道，在应用系统中位于最底层。遵循层次化原则设计底层驱动构件需要做到：

（1）针对应用场景和服务对象，分层组织构件。设计底层驱动构件的过程中，有一些与处理器相关的、描述了芯片寄存器映射的内容，这些是所有底层驱动构件都需要使用的，将这些内容组织成底层驱动构件的公共内容，作为底层驱动构件的基础。在底层驱动构件的基础上，还可使用高级的扩展构件调用底层驱动构件功能，从而实现更加复杂的服务。

（2）在构件的层次模型中，**上层构件可以调用下层构件提供的服务，同一层次的构件不存在相互依赖关系，不能相互调用**。例如，Flash 模块与 UART 模块是平级模块，不能在编写 Flash 构件时，调用 UART 驱动构件。即使要通过 UART 驱动构件函数的调用在 PC 屏幕上显示 Flash 构件测试信息，也不能在 Flash 构件内含有调用 UART 驱动构件函数的语句，应该编写上一层次的程序调用。平级构件是相互不可见的，只有深入理解这一点，并遵守之，才能更好地设计出规范的底层驱动构件。在操作系统下，平级构件不可见特性尤为重要。

2）易用性原则

易用性在于让调用者能够快速，理解地构件提供服务的功能并进行使用。遵循易用性原则设计底层驱动构件需要做到：**函数名简洁且达意；接口参数清晰，范围明确；使用说明语言精炼规范，避免二义性**。此外，在函数的实现方面，避免编写代码量过多。函数的代码量过多会难以理解与维护，并且容易出错。若一个函数的功能比较复杂，可将其“化整为零”，通过编写多个规模较小功能单一的子函数，再进行组合，实现最终的功能。

3）鲁棒性原则

鲁棒性在于为调用者提供安全的服务，避免在程序执行过程中出现异常状况。遵循鲁棒性原则设计底层驱动构件需要做到：**在明确函数输入输出的取值范围、提供清晰接口描述的同时，在函数实现的内部要有对输入参数的检测，对超出合法范围的输入参数进行必要的处理**；使用分支判断时，确保对分支条件判断的完整性，对缺省分支进行处理。例如，对 if 结构中的 else 分支和 switch 结构中的 default 安排合理的处理程序，**同时，不能忽视编译警告错误**。

4）内存可靠使用原则

对内存的可靠使用是保证系统安全、稳定运行的一个重要的考虑因素。遵循内存可靠使用原则设计底层驱动构件需要做到：

（1）优先使用静态分配内存。相比于人工参与的动态分配内存，静态分配内存由编译器维护，更为可靠。

（2）谨慎地使用变量。可以直接读写硬件寄存器时，不使用变量替代。避免使用变量暂存简单计算所产生的中间结果，使用变量暂存数据将会影响到数据的时效性。

（3）检测空指针。定义指针变量时必须初始化，防止产生“野指针”。

（4）检测缓冲区溢出，并为内存中的缓冲区预留不小于 20%的冗余。使用缓冲区时，对填充数据长度进行检测，不允许向缓冲区中填充超出容量的数据。

（5）对内存的使用情况进行评估。

5.3.2 编码风格基本规范

良好的编码风格能够提高程序代码的可读性和可维护性，而使用统一的编码风格在团队合作编写一系列程序代码时无疑能够提高集体的工作效率。本节给出了编码风格的基本规范，主要涉及文件、函数、变量、宏及结构体类型的命名规范；涉及空格与空行、缩进、断行等的排版规范；涉及文件头、函数头、行及边等的注释规范。

1．文件、函数、变量、宏及结构体类型的命名规范

命名的基本原则是：

（1）命名清晰明了，有明确含义，使用完整单词或约定俗成的缩写。通常，较短的单词可通过去掉元音字母形成缩写；较长的单词可取单词的头几个字母形成缩写，即“见名知意”。命名中若使用特殊约定或缩写，要有注释说明。

（2）命名风格要自始至终保持一致。

（3）为了代码复用，命名中应避免使用与具体项目相关的前缀。

（4）为了便于管理，对程序实体的命名要体现出所属构件的名称。

（5）使用英语命名。

（6）除宏命名外，名称字符串全部小写，以下划线“_”作为单词的分隔符，首尾字母不用“_”。

针对嵌入式底层驱动构件的设计需要，对文件、函数、变量、宏及数据结构类型的命令特别进行说明。

1）文件的命名

底层驱动构件在具体设计时分为两个文件，其中头文件命名为“<构件名>.h”，源文件命名为“<构件名>.c”，其中<构件名>表示具体的硬件模块的名称。例如，GPIO 驱动构件对应的两个文件为“gpio.h”和“gpio.c”。

2）函数的命名

底层驱动构件的函数从属于驱动构件，驱动函数的命名除要体现函数功能外，还需要使用命名前缀和后缀标识其所属的构件及不同的实现方式。**函数名前缀**：底层驱动构件中定义的所有函数均使用“<构件名>_”前缀表示其所属的驱动构件模块。例如，GPIO 驱动构件提供的服务接口函数命名为 gpio_init（初始化）、gpio_set（设定引脚状态）、gpio_get（获取引脚状态）等。**函数名后缀**：对同一服务的不同方式的实现，使用后缀加以区分，这样做的好处是：当使用底层构件组装软件系统时，避免构件之间出现同名现象，同时，名称要使人有“顾名思义”的效果。

3）函数形参变量与函数内局部变量的命名

对嵌入式底层驱动构件进行编码的过程中，需要考虑对底层驱动函数形参变量及驱动函数内部局部变量的命名。**函数形参变量**：函数形参变量名是使用函数时理解形参的最直观印象，表示传参的功能说明。特别地，若传入底层驱动函数接口的参数是指针类型，则在命名时应使用“_ptr”后缀加以标识。**局部变量**：局部变量的命名与函数形参变量类似。但函数形参变量名一般不取单个字符（如 i、j、k）进行命名，而 i、j、k 作为局部循环变量是允许的。这是因为，变量，尤其是局部变量，如果用单个字符表示，很容易写错（如 i 写成 j），而编译时又检查不出来，有可能为了这个小小的错误而花费大量的查错时间。

4）宏常量及宏函数的命名

宏常量及宏函数的命名全部使用大写字符，使用下划线“_”为分隔符。例如，在构件公共要素中定义的开关中断的宏如下。

```
#define ENABLE_INTERRUPTSasm(" CPSIE i")          //开总中断
#define DISABLE_INTERRUPTS asm(" CPSID i")        //关总中断
```

5）结构体类型的命名、类型定与变量声明

（1）结构体类型名称使用小写字母命名（<defined_struct_name>），定义结构体类型变量时，全部使用大写字母命名（<DEFINED_STRUCT_NAME>）。

（2）对结构体内部字段全部使用大写字母命名（< ELEM_NAME >）。

（3）定义类型时，同时声明一个结构体变量和结构体指针变量。

模板如下。

```
typedef  struct<defined_struct_name>
{
    <elem_type_1><ELEM_NAME_1>;    //对字段 1 含义的说明
    <elem_type_2><ELEM_NAME_2>;    //对字段 2 含义的说明
    ...
}  <DEFINED_STRUCT_NAME>, * <DEFINED_STRUCT_NAME_PTR>;
```

例如，当要定义一个描述 UART 设备初始化参数的结构体类型时，可有如下定义。

```
typedef  struct  uart_init
{
   uint_8 DEV_ID:                 //串口设备号
   uint_32 BUS_CLK;               //串口模块时钟频率
   uint_32 BAUD_RATE:             //串口通信波特率
}  UART_INIT_STRUCT,  * UART_INIT_PTR;
```

这样“uart_init”就是一种结构体类型，而 UART_INIT_STRUCT 是一个 uart_init 类型变量，UART_INIT_PTR 是 uart_init 类型指针变量。

2．排版

对程序进行排版是指，通过插入空格与空行，使用缩进、断行等手段，调整代码的书面版式，**使代码整体美观、清晰，从而提高代码的可读性**。

1）空行与空格

关于空行：相对独立的程序块之间须加空行。

关于空格：在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格，必要时加两个空格；进行非对等操作时，如果是关系密切的立即操作符（如->），其后不应加空格。采用这种松散方式编写代码的目的是使代码更加清晰。例如，只在逗号、分号后面加空格；在比较操作符，赋值操作符“=”、“+=”，算术操作符“+”、“%”，逻辑操作符“&&”，位域操作符“<<”、“^”等双目操作符的前后加空格；在“!”、“～”、“++”、“—”、“&”（地址运算符）等单目操作符前后不加空格；在“->”、“.”前后不加空格；在 if、for、while、switch 等与后面括号间加空格，使关键字更为突出、明显。

2）缩进

使用空格缩进，不使用 Tab 键，这样代码复制打印是不会造成错乱。代码的每一级均往右缩进 4 个空格的位置。函数或过程的开始、结构的定义及循环、判断等语句中的代码都要采用缩进风格，case 语句下的情况处理语句也要遵从语句缩进要求。

3）断行

（1）**较长的语句（>78 字符）要分成多行书写**，长表达式要在低优先级操作符处划分新行，操作符放在新行之首，划分出的新行要进行适当的缩进，使排版整齐，语句可读。

（2）循环、判断等语句中若有较长的表达式或语句，则要进行适应的划分，长表达式要在低优先级操作符处划分新行，操作符放在新行之首。

（3）若函数或过程中的参数较长，则要进行适当的划分。

（4）不允许把多个短语句写在一行中，即一行只写一条语句。特殊情况可用，例如“if (x>3)　x=3;”可以在一行。

（5）if、for、do、while、case、switch、default 等语句后的程序块分界符（如 C/C++语言的大括号‘{’和‘}’）应各独占一行并且位于同一列，且与以上保留字左对齐。

3．注释

在程序代码中使用注释，有助于对程序的阅读理解，说明程序在“做什么”，解释代码的目的、功能和采用的方法。编写注释时要注意：

（1）一般情况源程序有效注释量在 30%左右。

（2）注释语言必须准确、易懂、简洁。

（3）编写和修改代码的同时，处理好相应的注释。

（4）C 语言中采用“//”注释，不使用段注释“ /**/ ”。段注释用于调试，便于注释不用的代码。

为规范嵌入式底层驱动构件的注释，特别对文件头注释、函数头注释、行注释与边注释进行特别说明。

1）文件头注释

底层驱动构件的接口头文件和实现源文件的开始位置，使用文件头注释，如：

```
//=====================================================================
//文件名称：gpio.h
//功能概要：GPIO 底层驱动构件头文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-06-05  V2.0
//芯片类型：KEA128
//=====================================================================
```

2）函数头注释

在驱动函数的接口声明和函数实现前，使用函数头注释详细说明驱动函数提供的服务。在构件的头文件中必须添加完整的函数头注释，为构件使用者提供充分的使用信息。构件的源文件对用户是透明的，因此，在必要时可适当简化函数头注释的内容。例如：

```
//=====================================================================
//函数名称：gpio_init
//函数返回：无
//参数说明：port_pin：(端口号)|(引脚号)（例：PORTB|5 表示为 B 口 5 号脚）
//          dir：引脚方向（0=输入，1=输出,可用引脚方向宏定义）
//          state：端口引脚初始状态（0=低电平，1=高电平）
//功能概要：初始化指定端口引脚作为 GPIO 引脚功能，并定义为输入或输出，若是输出，
//          还指定初始状态是低电平或高电平
//=====================================================================
```

3）整行注释与边注释

整行注释文字，主要是对到下一个整行注释之前的代码进行功能概括与说明。边注释位于一行程序的尾端，对本语句或至下一边注释之间的语句进行功能概括与说明。此外，分支语句（条件分支、循环语句等）须在结束的“}”右方应边注释，表明该程序块结束的标记“end_……”，尤其在多重嵌套时。对于有特别含义的变量、常量，如果其命名不是充分自注释的，在声明时都必须加以注释，说明其含义。变量、常量、宏的注释应放在其上方相邻位置（行注释）或右方（边注释）。

5.3.3　公共要素文件

为某一款芯片编写驱动构件时，不同的构件存在公共使用的内容，将这些内容以构件的形式组织起来，成为构件公共要素。构件公共要素在底层驱动构件的体系中有着特殊的地位，为设备底层驱动构件的编写提供最基本的支持。在不同的应用环境间移植驱动构件时，都应根据软硬件的基本情况在构件公共要素文件中进行相关的配置，满足所有底层驱动构件正常工作时所需的基本和公共需求。所有底层驱动构件都包含对构件公共要素的引用，构件公共要素文件放在工程文件夹的“\Common”文件夹下，名为 common.h。本节以 common.h 文件内容为主线，介绍构件公共要素提供的服务。

1．芯片寄存器映射文件

每个底层驱动构件都是以硬件模块的特殊功能寄存器为操作对象的，因此，在 common.h 文件中包含了描述芯片寄存器映射的头文件，当底层驱动构件引用 common.h 文件时，即可使用片内寄存器映射文件中定义访问各自相关的特殊功能寄存器。

除包含芯片片内寄存器映像文件，还需要将内核及芯片相关文件引用到公共要素中。这些文件的功能简介见 4.4.4 节内容。一般地，开关总中断是嵌入式编程中常用的功能，当运行某写程序不希望被外部事件打断时，就可以暂时关闭中断系统，为程序执行提供一个“安静”的运行环境。开关总中断是嵌入式编程中常用的功能，C 语言的编译器无法为具体的芯片生成开关总中断的语句。

```
#define ENABLE_INTERRUPTS   __enable_irq       //开总中断
#define DISABLE_INTERRUPTS  __disable_irq      //关总中断
```

在 core_cmFunc.h 文件中可以看出，函数__enable_irq 和__disable_irq 分别是内嵌汇编的方式定义开关中断的语句，所以开关总中断的宏定义语句等同于以下语句。

```
#define ENABLE_INTERRUPTS  asm(" CPSIE i");    //开总中断
#define DISABLE_INTERRUPTS  asm(" CPSID i");   //关总中断
```

2．位操作宏函数

将编程时经常用到的寄存器位操作，即对寄存器的置位、清位及获得寄存器某一位状态的操作，定义成宏函数。设置寄存器某一位为 1，称为置位；设置寄存器某一位为 0，称为清位。这在底层驱动编程时经常用到。置位与清位的基本原则是：当对寄存器的某一位进行置位或清位操作时，不能干扰该寄存器的其他位，否则，可能会出现意想不到的错误。

综合利用“<<”、“>>”、“|”、“&”、“～”等位运算符，可以实现置位与清位，且不影响其他位的功能。下面以 8 位寄存器为例进行说明，其方法适用于各种位数的寄存器。设 R 为 8 位寄存器，下面说明将 R 的某一位置位与清位，而不干预其他位的编程方法。

（1）置位。要将 R 的第 3 位置 1，其他位不变，可以这样做：R |= (1<<3)，其中“1<<3”的结果是“0b00001000”，R |= (1<<3)也就是 R=R|0b00001000，任何数和 0 相或不变，任何数和 1 相或为 1，这样达到对 R 的第 3 位置 1，但不影响其他位的目的。

（2）清位。要将 R 的第 2 位清 0，其他位不变，可以这样做：R &= ～(1<<2)，其中“～(1<<2)”的结果是“0b11111011”，R&=～(1<<2)也就是 R=R&0b11111011，任何数和 1 相与不变，任何数和 0 相与为 0，这样达到对 R 的第 2 位清 0，但不影响其他位的目的。

（3）获得某一位的状态。(R>>4) & 1，是获得 R 第 4 位的状态，“R>>4”是将 R 右移 4 位，将 R 的第 4 位移至第 0 位，即最后 1 位，再和 1 相与，也就是和 0b00000001 相与，保留 R 最后 1 位的值，以此得到第 4 位的状态值。

为了方便使用，把这种方法改为带参数的“宏函数”，并且简明定义，放在公共头文件（common.h）中。使用该“宏”的文件，可以包含“common.h”文件。

```
#define  BSET(bit,Register)  ((Register) | = (1<<(bit)))     //置Register的第bit位为1
#define  BCLR(bit,Register)  ((Register) & = ～(1<<(bit)))   //清Register的第bit位
#define  BGET(bit,Register)  (((Register) >> (bit)) & 1)     //取Register的第bit位状态
```

这样就可以使用 BSET、BCLR、BGET 这些容易理解与记忆的标识，进行寄存器的置位、清位及获得寄存器某一位状态的操作。

3．重定义基本数据类型

嵌入式程序设计与一般的程序设计有所不同，在嵌入式程序中打交道的大多数都是底层硬件的存储单元或是寄存器，所以在编写程序代码时，使用的基本数据类型多以 8 位、16 位、32 位数据长度为单位。不同的编译器为基本整型数据类型分配的位数存在不同，但在编写嵌入式程序时要明确使用变量的字长，因此，需根据具体编译器重新定义嵌入式基本数据类型。重新定义后，不仅书写方便，也有利于软件的移植。例如：

```
//重定义基本数据类型（类型别名宏定义）
typedef  unsigned char          uint_8;     //无符号 8 位数，字节
typedef  unsigned short int     uint_16;    //无符号 16 位数，字
typedef  unsigned long int      uint_32;    //无符号 32 位数，长字
typedef  char                   int_8;      //有符号 8 位数
typedef  short int              int_16;     //有符号 16 位数
typedef  int                    int_32;     //有符号 32 位数
//不优化类型
typedef  volatile uint_8        vuint_8;    //不优化无符号 8 位数，字节
typedef  volatile uint_16       vuint_16;   //不优化无符号 16 位数，字
typedef  volatile uint_32       vuint_32;   //不优化无符号 32 位数，长字
typedef  volatile int_8         vint_8;     //不优化有符号 8 位数
typedef  volatile int_16        vint_16;    //不优化有符号 16 位数
typedef  volatile int_32        vint_32;    //不优化有符号 32 位数
```

通常有一些数据类型不能进行优化处理。在此，对不优化数据类型的定义作特别说明。不优化数据类型的修饰关键字是 volatile，它用于通知编译器，对其后面所定义的变量不能随意进行优化，因此，编译器会安排该变量使用系统存储区的具体地址单元，编译后的程序每次需要存储或读取该变量时，都会直接访问该变量的地址。若没有 volatile 关键字，则编译器可能会暂时使用 CPU 寄存器来存储，以优化存储和读取，这样，CPU 寄存器和变量地址的内容很可能会出现不一致现象。对 MCU 的映像寄存器的操作就不能优化，否则，对 I/O 口的写入可能被“优化”写入到 CPU 内部寄存器中，就会乱套。常用的 volatile 变量使用场合有：设备的硬件寄存器、中断服务例程中访问到的非自动变量、操作系统环境下多线程应用中被几个任务共享的变量。

5.3.4　头文件的设计规范

头文件描述了构件的接口，用户通过头文件获取构件服务。在本节中，将对底层驱动构件头文件的内容的编写加以规范，从程序编码结构、包含文件的处理、宏定义及设计服务接口等方面进行说明。

1．编码框架

编写每个构件的头文件时，应使用“#ifndef… #define … #endif”的编码结构，防止对头文件的重复包含。例如，若定义 GPIO 驱动构件，在其头文件 gpio.h 中，应有：

```
#ifndef GPIO_H
#define GPIO_H
```

```
    ……//文件内容
#endif
```

2．包含文件

包含文件命令为#include ，包含文件的语句统一安排在构件的头文件中，而在相应构件的源文件中仅包含本构件的头文件。将包含文件的语句统一置于构件的头文件中，使文件间的引用关系能够更加清晰地呈现。

3．使用宏定义

宏定义命令为#define，使用宏定义可以替换代码内容，替换内容可以是常数、字符串，甚至还可以是带参数的函数。利用宏定义的替换特性，当需要变更程序的宏常量或宏函数时，只需一次性修改宏定义的内容，程序中每个出现宏常量或宏函数的地方均会自动更新。

（1）使用宏定义表示构件中的常量，为常量值提供有意义的别名。例如，在 Light 构件（指示灯构件）中使用 GPIO 驱动构件，灯的亮暗状态与对应 GPIO 引脚高低电平的对应关系需根据外接电路而定，此时，将表示灯状态的电平信号值用宏常量的方式定义。当使用的外部电路发生变化时，对应地，仅需在 Light 构件中对表示灯亮暗状态的宏常量定义做适当变更，就可实现 Light 构件在新应用环境上的移植。

```
#define LIGHT_ON    0       //灯亮
#define LIGHT_OFF   1       //灯灭
```

（2）使用宏函数实现构件对外部请求服务的接口映射。在设计构件时，有时会需要应用环境为构件的基本活动提供服务。此时，采用宏函数表示构件对外部请求服务的接口，在构件中不关心请求服务的实现方式，这就为构件在不同应用环境下的移植提供了较强的灵活性。

4．设计服务接口

底层驱动构件通过服务函数接口提供完备而易用的服务，对外提供的服务接口在接口头文件应做声明。按照编码风格基本规范为驱动函数接口添加函数头注释。通过对设备驱动功能进行研究可以发现，对设备底层的操作一般可归为初始化、向设备发送信息、从设备读取信息和调整设备工作属性等，开关底层硬件模块中断也是底层驱动构件提供的常见服务。在此基础上构思底层驱动构件能够对外提供的服务，从而设计相应的服务接口（如函数名、传参列表及返回值）。设计驱动服务接口，要考虑为用户提供最简单的使用方式，提供最常用的配置参数，便于用户使用。

5.3.5 源程序文件的设计规范

编写底层驱动构件实现源文件基本要求，是实现构件通过服务接口对外提供全部服务的功能。为确保构件工作的独立性，实现构件高内聚、低耦合的设计要求，将构件的实现内容封装在源文件内部。对于底层驱动构件的调用者而言，通过服务接口获取服务，而不需要了解驱动构件提供服务的具体执行细节。因此，功能实现和封装是编写底层驱动构件实现源文件的主要考虑内容。

1．源程序文件中的#include

底层驱动构件的源文件（.c）中，只允许一处使用#include 包含自身头文件。需要包含的内容需在自身构件的头文件中包含，以便有统一、清晰的程序结构。

2．合理设计对外接口函数与内部函数

内部函数与外部函数要设计合理，函数参数个数及类型要考虑全面。外部函数是对外接口函数，供上层应用程序调用。在定义外部函数时，应该对函数名、函数功能、入口参数、函数返回值、使用说明、函数适用范围等进行详细描述，以增强程序的可读性。上层应用程序不能直接对构件的属性进行读取或设置，必须借助于该构件提供的接口操作函数来实现。在构件中的封装比较复杂功能的函数时，代码量不宜过长，此时，就应当将其中功能相对独立的部分封装成子函数。这些子函数仅在构件内部使用，不提供对外服务，因此被称为“内部函数”。为将内部函数的访问范围限制在构件的源文件内部，在创建内部函数时，应使用 static 关键字作为修饰符。

一般地，实现底层驱动构件的功能，需要和芯片片内模块的特殊功能寄存器打交道，通过对相应寄存器的配置实现对设备的驱动。某些配置过程对配置的先后顺序和时序有特殊要求，在编写驱动程序时要特别注意。

3．不使用全局变量

全局变量的作用范围可以扩大到整个应用程序，其中存放的内容在应用程序的任何一处都可以随意修改，一般可用于在不同程序单元间传递数据。但是，若在底层驱动构件中使用全局变量，其他程序即使不通过构件提供的接口也可以访问到构件内部，这无疑对构件的正常工作带来隐患。从软件工程理论中对封装特性的要求上看，也不利于构件设计高内聚、低耦合的要求。因此，在编写驱动构件程序时，**严禁使用全局变量**。用户与构件交互只能通过服务接口进行，即所有的数据传递都要通过函数的形参来接收，而不是使用全局变量。

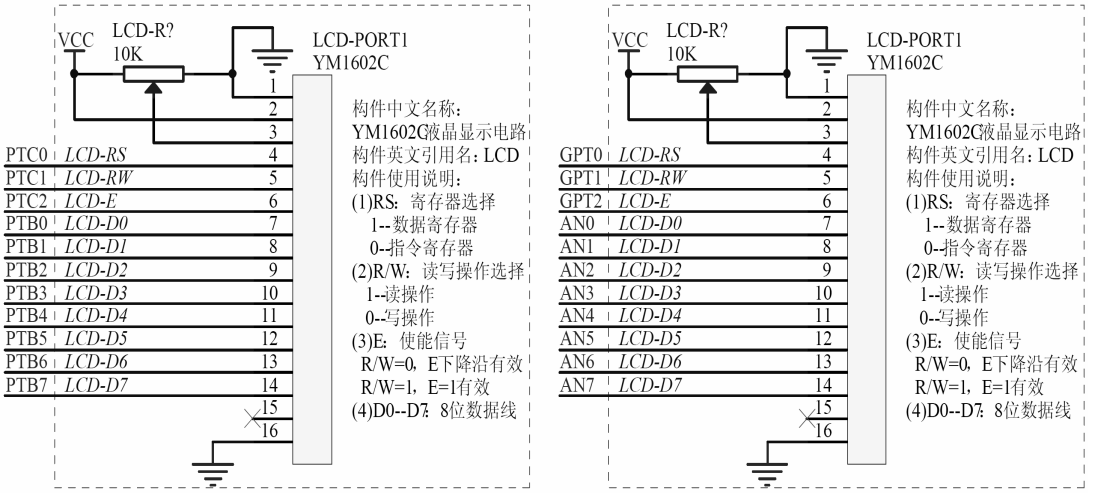
5.4　硬件构件及底层软件构件的重用与移植方法

重用是指在一个系统中，同一构件可被重复使用多次。移植是指将一个系统中使用到的构件应用到另外一个系统中。

1．硬件构件的重用与移植

对于以单 MCU 为核心的嵌入式应用系统而言，当用硬件构件“组装”硬件系统时，核心构件（即最小系统）有且只有一个，而中间构件和终端构件可有多个，并且相同类型的构件可出现多次。下面以终端构件 LCD 为例，介绍硬件构件的移植方法。

在应用系统 A 中，若 LCD 的数据线（LCD-D0～LCD-D7）与芯片 AW60（8 位 MCU）芯片的通用 IO 口的 B 口相连，C 口作为 LCD 的控制信号传送口，其中，LCD 寄存器选择信号 LCD-RS 与 C 口第 0 引脚连接，读写信号 LCD-RW 与 C 口第 1 引脚连接，使能信号 LCD_E 与 C 口第 2 引脚连接，则 LCD 硬件构件实例如图 5-6(a)所示。虚线框左边的文字（如 PTC0、PTC1 等）为接口网标，虚线框右边的文字（如 LCD-RS、LCD-RW 等）为接口注释。



(a) LCD 构件在系统 A 中的应用　　(b) LCD 构件在系统 B 中的应用

图 5-6　LCD 构件在实际系统中的应用

在应用系统 B 中，若 LCD 的数据线（LCD-D0～LCD-D7）与 MCF52233（32 位 MCU）芯片的通用 IO 口的 AN 口相连，TA 口的第 0、1、2 引脚分别作为寄存器选择信号 LCD-RS、读写信号 LCD-RW、使能信号 LCD_E，则 LCD 硬件构件实例如图 5-6（b）所示。

2．底层构件的移植

当一个已设计好的底层构件移植到另外一个嵌入式系统中时，其头文件和程序文件是否需要改动呢？这要视具体情况而定。例如，系统的核心构件发生改变（即 MCU 型号改变）时，底层内部构件头文件和某些对外接口函数也要随之改变，如模块初始化函数。

而对于外接硬件构件，希望不改动程序文件，而只改动头文件，那么，头文件就必须充分设计。以 LCD 构件为例，与图 5-6(a)相对应的底层构件头文件 lcd.h 可如下编写。

```
//=============================================================================
//文件名称：lcd.h
//功能概要：lcd 构件头文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2013-03-17    V1.2
//=============================================================================

#ifndef LCD_H
#define LCD_H

#include "common.h"
#include "gpio.h"

#define LCDRS       PORTC |(0)          //LCD 寄存器选择信号
#define LCDRW       PORTC |(1)          //LCD 读写信号
#define LCDE        PORTC |(2)          //LCD 读写信号
//LCD 数据引脚
```

```
#define LCD_D7      PORTB |(7)
#define LCD_D6      PORTB |(6)
#define LCD_D5      PORTB |(5)
#define LCD_D4      PORTB |(4)
#define LCD_D3      PORTB |(3)
#define LCD_D2      PORTB |(2)
#define LCD_D1      PORTB |(1)
#define LCD_D0      PORTB |(0)

//=============================================================================
//函数名称：LCDInit
//函数返回：无
//参数说明：无
//功能概要：LCD 初始化。
//=============================================================================
void LCDInit();
//=============================================================================
//函数名称：LCDShow
//函数返回：无
//参数说明：data[32]：需要显示的数组
//功能概要：LCD 显示数组的内容。
//=============================================================================
void LCDShow(uint_8 data[32]);

#endif
```

当 LCD 硬件构件发生如图 5-6（b）所示的移植时，显示数据传送口和控制信号传送口发生了改变，只需修改头文件，而不需修改 lcd.c 文件。

必须申明的是，本书给出构件化设计方法的目的是，在进行软硬件移植时，设计人员所做的改动要尽量小，而不是不做任何改动。希望改动尽可能在头文件中进行，而不希望改动程序文件。

第 6 章

串行通信模块及第一个中断程序结构

本章导读：本章阐述 KEA128 的串行通信模块构件化编程。主要内容有：6.1 节在简要阐述 UART 编程要素基础上，给出 KEA128 芯片 UART 引脚复用及技术要点，给出 UART 驱动构件接口函数说明及使用方法举例，这样就可以对 KEA128 芯片 UART 引脚进行编程操作了；6.2 节给出 UART 接收中断程序实例并给出 printf 的设置方法与使用；6.3 节给出 UART 驱动构件的制作方法。

本章参考资料：本章 6.1.3 节（KEA128 芯片 UART 引脚）参考自《KEA128 参考手册》第 10 和 12 章，6.3.1 节（UART 模块编程结构）参考自《KEA128 参考手册》第 33 章。



6.1 KEA128 芯片 UART 驱动构件及使用方法

串行通信是 MCU 与外界通信的最简便方式之一，也是嵌入式开发中重要的打桩测试手段。实现串行通信功能的硬件模块在一部分 MCU 中被称为通用异步收发器（Universal Asynchronous Receiver Transmitters，UART），在另一些 MCU 中被称为串行通信接口（Serial Communication Interface，SCI），通俗地说，就是“串口”。

6.1.1 UART 的基础知识要素

1. UART 硬件

在嵌入式 MCU 中，一般串行通信只需要三根线，分别称为发送线（TxD）、接收线（RxD）和地线 GND）。由于 UART 是用电平高低直接表达逻辑 1 与逻辑 0，因此有公共地（GND）。

MCU 引脚的 UART 模块引脚直接提供 TTL 电平信号。现在，一般笔记本电脑或 PC 没有串行通信接口，而有几个 USB 接口，在市场上可很容易的购买到 TTL-USB 到串行口转接器，以便进行嵌入式程序调试。

2. UART 的通信格式

串行通信的特点是：数据以字节为单位，按位的顺序（例如最高位优先）从一条传输线上发送出去。

在 MCU 的英文芯片手册上，通常说的异步串行通信采用的是 NRZ 数据格式，英文全称是：“standard non-return-zero mark/space data format”，可以译为：“标准不归零传号/空号数据格式”。这是一个通信术语，“不归零”的最初含义是：用负电平表示一种二进制值，正电平表示另一种二进制值，不使用零电平。“mark/space”即“传号/空号”分别是表示两种状态的

物理名称，逻辑名称记为“1/0”。对学习嵌入式应用的读者而言，只要理解这种格式只有“1”、“0”两种逻辑值就可以了。图 6-1 给出了 8 位数据、无校验情况的传送格式。

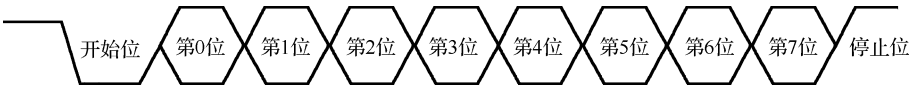


图 6-1 串行通信数据格式

这种格式的空闲状态为“1”，发送器通过发送一个“0”表示一个字节传输的开始，随后是数据位（在 MCU 中一般是 8 位或 9 位，可以包含校验位）。最后，发送器发送 1 到 2 位的停止位，表示一个字节传送结束。若继续发送下一字节，则重新发送开始位[①]，开始一个新的字节传送。若不发送新的字节，则维持“1”的状态，使发送数据线处于空闲。从开始位到停止位结束的时间间隔称为一帧（Frame)。所以，也称这种格式为帧格式。每发送一个字节，都要发送“开始位”与“停止位”，这是影响异步串行通信传送速度的因素之一。

3．串行通信的波特率

位长（Bit Length）也称为位的持续时间（Bit Duration），其倒数就是单位时间内传送的位数。人们把每秒内传送的位数叫做波特率（Baud Rate）。波特率的单位是位/秒，记为 bps。bps 是英文 bit per second 的缩写，习惯上这个缩写不用大写，而用小写。通常情况下，波特率的单位可以省略。

通常使用的波特率有 1200、1800、2400、4800、9600、19200、38400、57600 和 115200 等。在包含开始位与停止位的情况下，发送一个字节需 10 位，很容易计算出，在各波特率下，发送 1 KB 所需的时间。显然，这个速度相对于目前许多通信方式而言是慢的，那么，异步串行通信的速度能否提得很高呢？答案是不能。因为随着波特率的提高，位长变小，以至于很容易受到电磁源的干扰，通信就不可靠了。当然，还有通信距离问题，距离小，可以适当提高波特率，但这样毕竟提高的幅度非常有限，达不到大幅度提高的目的。

6.1.2 UART 驱动构件封装要点分析

UART 具有初始化、接收和发送三种基本操作。按照构件的思想，可将它们封装成三个独立的功能函数，初始化函数完成对 UART 模块的工作属性的设定，接收和发送功能函数则完成实际的通信任务。对 UART 模块进行编程，实际上已经涉及对硬件底层寄存器的直接操作，因此，可将初始化、接收和发送三种基本操作所对应的功能函数共同放置在命名为 uart.c 的文件中，并按照相对严格的构件设计原则对其进行封装，同时配以命名为 uart.h 的头文件，用来定义模块的基本信息和对外接口。

按照模块所具有的基本操作来确定构件中应该具有哪些功能集合，是很自然也很重要的事情。但是，要实现编程的构件化，对具体的函数原型的设计则是重中之重，函数原型设计的好坏直接影响构件化编程的成败。下面就以 UART 的初始化、接收和发送三种基本操作为例，来说明实现构件化编程的全过程。

① 这就是“异步”的含义。

需要说明的是，实现构件化编程的 UART 软件模块应当具有以下几个特点。

（1）UART 模块是最底层的构件，它主要向上提供三种服务，分别是 UART 模块的初始化、接收单个字节和发送单个字节，向下则直接访问模块寄存器，实现对硬件的直接操作。另外，从实际使用角度出发，它还需要封装接收 *N* 个字节和发送 *N* 个字节的功能函数。

（2）UART 模块在软件上对应 1 个 uart.c 程序源代码文件和 1 个 uart.h 头文件，当需要使用 UART 驱动构件时，大多数情况下只需简单拷贝这两个文件即可，无须对源代码文件和头文件进行修改，只有当进行不同芯片之间的移植时，才需要修改头文件中与硬件相关的宏定义。

（3）上层构件或软件在使用该构件时，严禁通过全局变量来传递参数，所有的数据传递都直接通过函数的形式参数来接收。这样做不但使得接口简洁，更可以避免全局变量可能引发的安全隐患。

通过以上分析，可以设计 UART 驱动构件的 8 个基本功能函数。

（1）初始化：uart_init (uint_8 uartNo,uint_32 sel_clk,uint_32 baud_rate);

（2）发送单个字节：uint_8 uart_send1(uint_8 uartNo, uint_8 ch);

（3）发送 *N* 个字节：uint_8 uart_sendN (uint_8 uartNo ,uint_16 len ,uint_8* buff);

（4）发送字符串：uint_8 uart_send_string(uint_8 uartNo, void *buff);

（5）接收单个字节：uint_8 uart_re1 (uint_8 uartNo,uint_8 *fp);

（6）接收 *N* 个字节：uint_8 uart_reN (uint_8 uartNo ,uint_16 len ,uint_8* buff);

（7）使能串口接收中断：uart_enable_re_int(uint_8uartNo);

（8）禁止串口接收中断：uart_disable_re_int(uint_8uartNo)。

6.1.3　KEA128 芯片 UART 引脚

KEA128 中共有 3 个 UART 模块，分别为 UART0、UART1 和 UART2。每个 UART 的发送数据引脚为 UARTx_TX，接收数据引脚为 UARTx_RX。“x”表示串口模块编号，取值为 0～2。它们并不是固定在哪几个引脚上，而是通过系统集成模块（System Integration Module，SIM）提供的引脚选择寄存器编程来设定。根据附录 A（SKEAZ128MLK 引脚功能分配），可以配置为串口的引脚见表 6-1。

表 6-1　KEA128 的串口引脚及默认使用的引脚

引脚号	引脚名	ALT2	ALT3	ALT4
16	PTI1	IRQ	UART2_TX	
17	PTI0	IRQ	UART2_RX	
26	PTD7	UART2_TX		
27	PTD6	UART2_RX		
41	PTB1	UART0_TX		ADC0_SE5
42	PTB0	UART0_RX	PWT_IN1	ADC0_SE4

续表

引脚号	引脚名	ALT2	ALT3	ALT4
43	PTF3	UART1_TX		
44	PTF2	UART1_RX		
59	PTA3	UART0_RX	I2C0_SCL	
60	PTA2	UART0_TX	I2C0_SDA	
63	PTC7	UART1_TX		
64	PTC6	UART1_RX		

6.1.4 KEA128 芯片 UART 驱动构件头文件及使用方法

头文件 uart.h 中的内容可分为两个主要的部分，分别是 8 个函数原型的声明和外设模块寄存器相关信息的定义。前者给出了本 UART 驱动构件对上层构件或软件所提供的接口函数，而后者则指明了本“元构件[①]”与具体硬件相关的信息。串行通信头文件 uart.h 含有串行通信寄存器和标志位定义以及串行通信相关函数声明，下面简要给出了它的主要内容。以初始化函数 uart_init 为例，通道号、当前的系统时钟、希望实现的通信波特率以及初始化时是否使能中断，都被设计为函数参数。这样的话，应用程序和上层构件在使用（调用）它时，将具有极大的灵活性。文件还给出了必要的硬件相关信息，当要把该构件移植到其他芯片时，就必须检查并修改这些信息。

1. UART 驱动构件的头文件（uart.h）

```
//=============================================================================
//文件名称：uart.h
//功能概要：UART 底层驱动构件头文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-06-05  V2.0
//芯片类型：KEA128
//=============================================================================
#ifndef _UART_H                 //防止重复定义（开头）
#define _UART_H

#include "common.h"             //包含公共要素头文件

//宏定义串口号
#define UART_0    0
#define UART_1    1
#define UART_2    2

//宏定义串口使用的引脚（由具体硬件板决定）
#define UART_0_GROUP   2        //UART_0: 1=PTA3～2 脚,2=PTB1～0 脚
#define UART_1_GROUP   1        //UART_1: 1=PTC7～6 脚,2=PTF3～PTF2 脚
#define UART_2_GROUP   1        //UART_2: 1=PTD7～6 脚,2=PTI1～0 脚
```

① UART 模块是最基本最底层的构件，它在满足功能完整性的情况下，具有不可再分的特性，可以把它叫做“元构件”。

```
//=========================================================================
//函数名称：uart_init
//功能概要：使用总线时钟（BUS_CLK_KHZ），初始化 uart 模块
//参数说明：uartNo:串口号 UART_0、UART_1、UART_2
//         baud:波特率 1200、2400、4800、9600、19200、115200
//函数返回：无
//=========================================================================
void uart_init(uint_8 uartNo, uint_32 baud_rate);

//=========================================================================
//函数名称：uart_send1
//参数说明：uartNo: 串口号 UART_0、UART_1、UART_2
//         data:要发送的字节
//函数返回：函数执行状态，0=正常；1=异常。
//功能概要：串行发送 1 个字节
//=========================================================================
uint_8 uart_send1(uint_8 uartNo, uint_8 data);

//=========================================================================
//函数名称：uart_sendN
//参数说明：uartNo: 串口号 UART_0、UART_1、UART_2
//         buff: 发送缓冲区
//         len:发送字节数
//函数返回：函数执行状态，0=正常；1=异常。
//功能概要：串行发送多个字节
//=========================================================================
uint_8 uart_sendN(uint_8 uartNo, uint_16 len, uint_8* buff);

//=========================================================================
//函数名称：uart_send_string
//参数说明：uartNo:UART 模块号 UART_0、UART_1、UART_2
//         buff:要发送的字符串的首地址
//函数返回：函数执行状态，0=正常；1=异常。
//功能概要：从指定 UART 端口发送一个以'\0'结束的字符串(以双引号表达的字符串，即以'\0'
//         结束的字符串）
//例：uart_send_string(UART_0,"abcdefg"); 即可发送字符  abcdefg
//=========================================================================
uint_8 uart_send_string(uint_8 uartNo, uint_8 *buff);

//=========================================================================
//函数名称：uart_re1
//参数说明：uartNo: 串口号 UART_0、UART_1、UART_2
//         fp:接收成功标志的指针，*fp=0，成功接收；*fp=1，接收失败
//函数返回：接收返回字节
//功能概要：串行接收 1 个字节
//=========================================================================
uint_8 uart_re1(uint_8 uartNo, uint_8 *fp);
```

```
//=============================================================================
//函数名称：uart_reN
//参数说明：uartNo：串口号 UART_0、UART_1、UART_2
//          buff：接收缓冲区
//          len:接收长度
//函数返回：函数执行状态，0=正常;1=异常
//功能概要：串行接收 n 个字节
//=============================================================================
uint_8 uart_reN(uint_8 uartNo, uint_16 len, uint_8* buff);

//=============================================================================
//函数名称：uart_enable_re_int
//参数说明：uartNo：串口号:UART_0、UART_1、UART_2
//函数返回：无
//功能概要：开串口接收中断
//=============================================================================
void uart_enable_re_int(uint_8 uartNo);

//=============================================================================
//函数名称：uart_disable_re_int
//参数说明：uartNo：串口号 UART_0、UART_1、UART_2
//函数返回：无
//功能概要：关串口接收中断
//=============================================================================
void uart_disable_re_int(uint_8 uartNo);
#endif                          //防止重复定义（结尾）
```

2．UART 驱动构件的使用方法

在 UART 驱动构件的头文件（uart.h）中包含的内容有：用宏定义方式统一了使用的串口号（UART_0、UART_1、UART_2）；给出 8 个对外服务函数的接口说明及声明，函数包括串口初始化函数（uart_init）、串行发送 1 个字节（uart_send1）、串行发送 *n* 个字节（uart_sendN）、发送一个以'\0'结束的字符串（uart_sendN）、串行接收 1 个字节（uart_re1）、串行接收 *n* 个字节（uart_reN）、开接收中断（uart_enable_re_int）和关接收中断（uart_disable_re_int）。

宏定义 UART_x_GROUP（x=0～2）方便确定和改变每个 UARTx_TX 和 UARTx_RX 映射到的引脚，例如，“#define UART_2_GROUP 2”，表明 UART2_TX 和 UART2_RX 引脚分别映射到引脚 PTI1 和 PTI0 上。

现在，以控制 UART_2 发送和接收数据为例，假设发送和接收的对应引脚为 PTD7 和 PTD6，介绍构件使用方法，使用步骤举例如下。

（1）修改宏定义 UART_2_GROUP，确定发送和接收的引脚为 PTD7 和 PTD6。

```
//UART_2: 1=PTD7～6 脚,2=PTI1～0 脚
#define UART_2_GROUP1                    //UART_2 使用的硬件引脚 PTD7～6
```

（2）初始化串口 UART_2，时钟源为总线时钟，波特率使用 9600。

```
uart_init (UART_2,BUSCLK, 9600);        //波特率使用 9600
```

（3）使能串口 UART_2 接收中断，通过串口 2 中断服务例程接收数据。

```
uart_enable_re_int(UART_2);             //使能串口 2 接收中断
```

（4）串口 UART_2 发送字符串。

```
uart_send_string(UART_2, "Hello Uart! 2014-12-11\r\n"); //串口发送初始化提示
```



6.2　UART 接收中断程序实例

6.2.1　KEA128 的中断服务程序及其“注册”

1．中断服务程序设计

中断服务程序的设计与普通构件函数设计是一样的，只是这些程序只有在产生中断时才被执行。为了规范编程，将各个中断服务程序放在 isr.c 中断服务程序源文件。例如，编写一个串口中断服务程序，串口 1 有数据到来时产生中断，执行“isr_uart1_re(void)”接收中断服务程序。服务程序首先进入临界区[①]关总中断，接收一个到来的字符。若接收成功成功，则把这个字符发送回去，退出临界区。

```
//=============================================================================
//文件名称：isr.c
//功能概要：中断底层驱动构件源文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2014-12-10  V1.0
//芯片类型：KEA128
//=============================================================================

#include "isr.h"

//========================中断函数服务例程================================
//串口 0 接收中断服务例程
void isr_uart0_re(void)
{
    uint_8 ReData;
    uint_8 flag;

    DISABLE_INTERRUPTS;          //关总中断

    flag=1;
    g_uart_num=0;                //标记当前使用的串口号
```

① 有些情况下，一些程序段是需要连续执行而不能被打断的，此时，程序对 CPU 资源的使用是独占的，此时称为“临界状态”，不能被打断的过程称为对“临界区”的访问。为防止在执行关键操作时被外部事件打断，一般通过关中断的方式使程序访问临界区，屏蔽外部事件的影响。执行完关键操作后退出临界区，打开中断，恢复对中断的响应能力。

```
    ReData = uart_re1(UART_0, &flag);
    if (0 == flag)
    {
        uart_send1(UART_0, ReData);
    }

    ENABLE_INTERRUPTS;          //开总中断
}


//串口 1 接收中断服务例程
void isr_uart1_re(void)
{
    uint_8 ReData;
    uint_8 flag;

    DISABLE_INTERRUPTS;         //关总中断

    flag=1;
    g_uart_num=1;               //标记当前使用的串口号

    ReData = uart_re1(UART_1, &flag);
    if (0 == flag)
    {
        uart_send1(UART_1, ReData);
    }

    ENABLE_INTERRUPTS;          //开总中断
}

//串口 2 接收中断服务例程
void isr_uart2_re(void)
{
    uint_8 ReData;
    uint_8 flag;

    DISABLE_INTERRUPTS;         //关总中断

    flag=1;
    g_uart_num=2;               //标记当前使用的串口号

    ReData = uart_re1(UART_2, &flag);
    if (0 == flag)
    {
        uart_send1(UART_2, ReData);
    }

    ENABLE_INTERRUPTS;          //开总中断
}
```

2．中断服务程序的“注册”

中断服务程序的“注册”就是将用户编写好的中断服务程序入口的地址，添加到中断向量表中，这样就可将中断向量表的缺省表项替换为用户编写好的中断服务程序名。但是，中断向量表文件是工程框架的重要内容之一，对它的操作要特别小心，因此，在设计工程框架的时候，为了避免用户直接对中断向量表进行修改，而采用“弱定义”的方式为用户提供编程接口，既方便用户使用，同时也可提高系统编程的安全性。

在启动文件 startup_SKEAZ1284.S 中的中断向量表__isr_vector 里，已经为所有内核异常及外设中断的中断向量以弱符号[①]的方式设置了缺省值，用户只需在 isr.h 文件中定义与对应中断向量表项的缺省值同名的函数，再重新将用户自定义的中断服务例程函数指向此函数即可。例如，串口 1 接收中断服务程序 isr_uart1_re(void)，查看中断向量表中 UART1 的中断向量缺省值为 UART0_IRQHandler(void)，只要在 isr.h 文件中加入如下代码即可。

```
//2 注册串口 1 中断向量
void  UART1_IRQHandler(void);
#define  isr_uart1_re  UART1_IRQHandler
```

6.2.2　UART 接收中断程序实例

测试程序功能概述如下。

（1）串口通信格式：波特率 9600，1 位停止位，无校验。

（2）上电或按复位按钮时，调试串口输出“Hello Uart! 2015-06-05”。

（3）PC 向 MCU 发送数据时，MCU 进入串口接收中断，将接收的一个字节直接回发，三个串口具有相同功能。

（4）主循环中，改变 LIGHT_0～LIGHT_3 小灯状态（四盏灯同时闪烁），同时，发送当前使用的串口号。

测试工程见网上光盘“..\program\CH06-KEA128-UART”文件夹。MCU 方的主程序及中断服务例程如下。

1．main.c 文件内容

```
//说明见工程文件夹下的 Doc 文件夹内 Readme.txt 文件
#define GLOBLE_VAR      //只需在 main.c 中定义一次，用来防止全局变量的重复定义

#include "includes.h"

int main(void)
 {
   //1. 声明主函数使用的变量
   uint_32 mRuncount;                    //主循环计数器
   //2. 关总中断
```

① 弱符号可被同名强符号覆盖，C 语言中编译器默认函数和初始化了的全局变量为强符号。

```
    DISABLE_INTERRUPTS;                 //关总中断
    //3. 初始化外设模块
    uart_init (UART_Debug,9600);        //初始化"调试串口"
    uart_init (UART_0,9600);
    uart_init (UART_1,9600);
    uart_init (UART_2,9600);
    light_init(LIGHT_0, LIGHT_OFF);     //初始化灯 0
    light_init(LIGHT_1, LIGHT_OFF);     //初始化灯 1
    light_init(LIGHT_2, LIGHT_OFF);     //初始化灯 2
    light_init(LIGHT_3, LIGHT_OFF);     //初始化灯 3
printf("Hello Uart! 2015-06-05\r\n");  //串口发送初始化提示
    //4. main 中使用的局部变量初始化
    mRuncount = 0;                      //主循环计数器
    //5. 全局变量初始化
    g_uart_num=UART_Debug;              //标记当前使用的串口号（初值）
    //6. 使能模块中断
    uart_enable_re_int(UART_Debug);     //使能 UART_Debug 的接收中断
    uart_enable_re_int(UART_0);
    uart_enable_re_int(UART_1);
    uart_enable_re_int(UART_2);
    //7. 开总中断
    ENABLE_INTERRUPTS;   //开总中断

    //进入主循环
    //主循环开始=============================================================
    for (;;)
    {
        //运行指示灯闪烁-----------------------------------------------------
        mRuncount++;                          //主循环次数计数器+1
        if (mRuncount >= RUN_COUNTER_MAX)     //主循环次数计数器大于设定的宏常数
        {
            mRuncount = 0;                    //主循环次数计数器清零
            light_change(LIGHT_0);            //灯 0（LIGHT_0）状态变化
            light_change(LIGHT_1);            //灯 1（LIGHT_1）状态变化
            light_change(LIGHT_2);            //灯 2（LIGHT_2）状态变化
            light_change(LIGHT_3);            //灯 3（LIGHT_3）状态变化
            printf("UART_%d",g_uart_num);     //串口发送当前使用的串口号
        }
        //以下加入用户程序---------------------------------------------------
    }  //主循环 end_for

    //主循环结束=============================================================
    return 0;
}
```

2．isr.h 文件内容

```
//=========================================================================
//文件名称：isr.h
//功能概要：中断底层驱动构件头文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
```

```
//版本更新：2014-12-10  V1.0
//芯片类型：KEA128
//============================================================================

#include "includes.h"

#ifndef _ISR_H                  //防止重复定义（ISR_H  开头）
#define _ISR_H

//===============用户中断向量表注册表=====================
//1 注册串口 0 中断向量
void UART0_IRQHandler(void);
#define isr_uart0_re  UART0_IRQHandler

//2 注册串口 1 中断向量
void UART1_IRQHandler(void);
#define isr_uart1_re  UART1_IRQHandler

//3 注册串口 2 中断向量
void UART2_IRQHandler(void);
#define isr_uart2_re  UART2_IRQHandler

#endif                          //防止重复定义（ISR_H  结尾）
```

3．isr.c 文件内容

```
//============================================================================
//文件名称：isr.c
//功能概要：中断底层驱动构件源文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2014-12-10  V1.0
//芯片类型：KEA128
//============================================================================

#include "isr.h"

//========================中断函数服务例程================================
//串口 0 接收中断服务例程
void isr_uart0_re(void)
{
    uint_8 ReData;
    uint_8 flag;

    DISABLE_INTERRUPTS;             //关总中断

    flag=1;
    g_uart_num=0;                   //标记当前使用的串口号
```

```
    ReData = uart_re1(UART_0, &flag);
    if (0 == flag)
    {
        uart_send1(UART_0, ReData);
    }

    ENABLE_INTERRUPTS;              //开总中断
}


//串口 1 接收中断服务例程
void isr_uart1_re(void)
{
    uint_8 ReData;
    uint_8 flag;

    DISABLE_INTERRUPTS;             //关总中断

    flag=1;
    g_uart_num=1;                   //标记当前使用的串口号

    ReData = uart_re1(UART_1, &flag);
    if (0 == flag)
    {
        uart_send1(UART_1, ReData);
    }

    ENABLE_INTERRUPTS;              //开总中断
}

//串口 2 接收中断服务例程
void isr_uart2_re(void)
{
    uint_8 ReData;
    uint_8 flag;

    DISABLE_INTERRUPTS;             //关总中断

    flag=1;
    g_uart_num=2;                   //标记当前使用的串口号

    ReData = uart_re1(UART_2, &flag);
    if (0 == flag)
    {
        uart_send1(UART_2, ReData);
    }

    ENABLE_INTERRUPTS;              //开总中断
}
```

6.2.3　printf 的设置方法与使用

除了使用 UART 驱动构件中封装的 API 函数外，还可以使用格式化输出函数 printf 来灵活地从串口输出调试信息。printf 函数的实现在工程目录“..\07_App_Component\printf\printf.c”文件中，同文件夹下的 printf.h 头文件则包含了 printf 函数的声明，若要使用 printf 函数，可在工程的总头文件“..\08_Source\includes.h”中将 printf.h 包含进来，以便其他文件使用。

在使用 printf 函数之前，需要先进行相应的设置来将其与希望使用的串口模块关联起来。设置步骤如下。

（1）在 printf 头文件“..\ 07_App_Component\printf\printf.h”中宏定义需要与 printf 相关联的调试串口号。例如：

```
#define UART_Debug  UART_2    //printf 函数使用的串口号
```

（2）在 printf.c 文件中找到函数 voidout_char (char ch)，将其内容改为 UART 驱动构件中发送单个字符的函数，使用的 UART 模块号由发送单个字符函数的参数带入。例如：

```
void out_char (char ch)
{
    uart_send1(UART_Debug,ch);
}
```

（3）在使用 printf 前，调用 UART 驱动构件中的初始化函数对使用的调试串口进行初始化，配置其波特率。例如：

```
uart_init (UART_Debug,9600);            //初始化"调试串口"
```

这样就将相应的串口模块与 printf 函数关联起来了。

关于 printf 函数的使用方法，请参见附录 C 的介绍。

6.3　UART 驱动构件的制作方法

6.3.1　UART 模块编程结构

以下寄存器的用法在 KEA128 的芯片手册上有详细的说明，下面按初始化顺序阐述基本编程需要使用的寄存器。注意：下面所列寄存器名中的“x”表示 UART 模块编号，取 0～2。

1．寄存器地址分析

每个 UART 模块有其对应的寄存器。以下地址分析均为十六进制，为书写简化起见，在不至引起歧义的情况下，略去十六进制的前缀“0x”不写。

UART 模块 x 的寄存器的地址=4006_A000+x×1000+n×1（x=0～2；n=0～7，n 代表寄存器号）。

2．控制寄存器

1）UARTx 控制寄存器 2（UARTx_C2）

UARTx 控制寄存器 2（UARTx_C2）主要用于收/发及相关中断控制设置。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
复位	0							

D7（TIE）—发送中断使能位。与状态寄存器中的 TDRE 配合使用，TIE=0，发送中断禁用（使用轮询）；TIE=1，当 TDRE=1 时，发生中断请求。

D6（TCIE）—发送完成中断使能位。与状态寄存器中的 TC 位配合使用，TCIE=0，TC 对应的中断禁用（使用轮询）；TCIE=1，当 TC=1 时，发生中断请求。

D5（RIE）—接收中断使能位。与状态寄存器中的 RDRF 配合使用，RIE=0，RDRF 中断禁止（使用轮询）；RIE=1，当 RDRF=1 时，发生中断请求。

D4（ILIE）—空闲线中断使能位。与状态寄存器中的 IDLE 配合使用，ILIE=0，IDLE 中断禁止（使用轮询）；ILIE=1，当 IDLE=1 时，发生中断请求。

D3（TE）—发送器使能位。TE 必须是 1 来使用 UART 发送器。通常在 TE=1 时，UART_TX 引脚作为 UART 系统的输出。当 UART 配置为单线模式（LOOPS=1 且 RSRC=1）时，UART_C3 中的 TXDIR 位将控制单线模式下 UART_TX 引脚的通信方向。通过对 TE 位先写 TE=0 然后写 TE=1，TE 位也可以对空闲字符排队。当 TE 为 0 时，发送器一直控制的端口 UART_TX 引脚，直到完成任何数据、等待空闲或等待中止符的传输，然后才允许引脚为三态。TE=0，发送器禁止；TE=1，发送器使能。

D2（RE）—接收器使能。当 UART 接收器关闭或 LOOPS 被置位，UART 不使用 UART_RX 脚。当 RE 被写 0，接收机完成接收的当前字符（如有的话）。RE=0，接收器禁止；RE=1，接收器使能。

D1（RWU）—接收器唤醒控制位。向该位写 1 可将接收器设置为待机状态，等待对选中的条件进行自动检测。唤醒方式有空闲线唤醒（WAKE=0）和地址位唤醒（WAKE=1）2 种。RWU=0，正常 UART 接收器操作；RWU=1，接收器等待唤醒条件。

D0（SBK）—发送中止使能位。在发送数据流中，写一个 1 然后一个 0 到 SBK 队列的中止字符。如果 BRK13 =1 时，只要 SBK 被设定，其他中止字符中的 10 至 13，或 13 至 16，逻辑 0 的位时间排队。根据 SBK 的设置，清除当前正在发送的信息，在软件清除 SBK 前，第二个中止字符可能会排队。SBK=0，正常发送操作；SBK=1，队列中止字符被发送。

2）UARTx 控制寄存器 1（UARTx_C1）

UARTx 控制寄存器 1（UARTx_C1）主要用于设置 SCI 的工作方式，可选择运行模式、唤醒模式、空闲类型检测及奇偶校验等。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	LOOPS	UARTSWAI	RSRC	M	WAKE	ILT	PE	PT
复位	0							

D7（LOOPS）—循环模式选择。在循环模式和正常的 2 针全双工模式之间选择，当 LOOPS=1，则发送器的输出连接到接收器的输入，可用于循环模式或单线模式，单线模式下，UART 不使用 UART_RX 引脚（见 RSRC）；LOOPS=0，采用不同的引脚正常操作 UART_RX 和 UART_TX（全双工）。

D6（UARTSWAI）—休眠使能/UART 等待模式停止位。UARTSWAI=0，UART 时钟在等待模式下仍然运行，这样 UART 就可以作为唤醒 CPU 的中断源；UARTSWAI=1，当 CPU 处于等待模式时，UART 时钟停止。

D5（RSRC）—接收器信号源位。在 LOOPS 置为 1 时，该位有效；当 LOOPS 被置位，接收器的输入在内部连接 UART_TX 引脚，RSRC 决定这个连接是否也被连接到发送器的输出。RSRC=0，选择内部循环模式，UART 不使用 UART_RX 的引脚；RSRC=1，UART 采用单线模式，UART_TX 引脚被连接到发送器的输出和接收器的输入。

D4（M）—数据帧格式选择位（9 位或 8 位模式选择）。当 M=0 时，接收和发送使用 8 位数据字符；当 M=1 时接收和发送使用 9 位数据字符。

D3（WAKE）—接收器唤醒模式选择位。WAKE=0，空闲线唤醒；WAKE=1，地址标志唤醒。

D2（ILT）—空闲线类型选择位。ILT=1，在停止位后开始对空闲特征位计数，ILT=0，在开始位后立即对空闲特征位开始计数。

D1（PE）—奇偶校验使能位。PE=0，奇偶校验禁止；PE=1，奇偶校验使能。当奇偶校验使能，停止位的前一位被视为奇偶校验位。

D0（PT）—奇偶校验类型位。当 PE 使能（PE=1）时，PT=0 偶校验，PT=1 奇校验。

3．状态寄存器

UARTx_S1 寄存器为 UART 中断或 DMA 请求提供 MCU 的输入，这个寄存器可以由 MCU 进行轮询来检测，也可以通过读状态寄存器之后读或写（取决于中断标志类型）UART 数据寄存器来清除标志。其他的指令只要不影响 I/O 处理也可以插入到上述两步中执行，该寄存器只读，不能写。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF
复位	1	1	0	0	0	0	0	0

D7（TDRE）—发送数据寄存器空标志位。该位在复位之后置 1，或者当一个发送数据从缓冲区转移到发送移位器后，该位置位。为清除 TDRE，当 TDRE=1 时，应先对 UARTx_S1 中的 TDRE 进行读操作，然后写 UART 数据寄存器 UART_D。TDRE=0，发送数据寄存器（缓冲器）已满；TDRE=1，发送数据寄存器（缓冲器）为空。

D6（TC）—发送完成标志位。TC=0，正在发送；TC=1，发送完成。TC 位在复位后置为 1；或当 TDRE=1，且无数据、前导符或中止符正在发送，TC 置为 1。TC=1 时，可通过先读取 UARTx_S1，然后进行以下任意一种操作，TC 位将自动清除：①向 UARTx_D 寄存器写入数据；②通过向 TE 写 0，然后向 TE 写 1，对一个前导字符排队；③通过向控制寄存器 2 的 SBK 位写 1，对一个中止字符排队。

D5（RDRF）—接收数据寄存器已满标志位。当接收缓冲区满了以后，该位置位。注意：为清除 RDRF，应先对 UARTx_S1 中的 RDRF 进行读操作，然后读 UART 数据寄存器。RDRF=0，接收数据寄存器（缓冲器）空；RDRF=1，接收数据寄存器（缓冲器）已满。

D4（IDLE）—空闲线标志。如果 UART 接收线在活动周期之后的空闲持续一个字符时间，则该位置位。当控制寄存器 1 中的 ILT=0 时，接收器从开始位计时空闲位时间。因此，如果接收到的字符全为 1，那么这些位的时间加上停止位的时间是接收器检测空闲线的时间。当控制寄存器 1 中的 ITL=1 时，接收器从停止位开始计时空闲位时间，停止位和刚发送字符中的任意高电平位的时间不能作为接收器检测空闲线的时间。要清除该标志位，可向该位写 1。该位清除后，只有在接收到一个新的字符且 RDRF=1 时，IDLE 才能再次置位，即使接收线在额外的周期内保持空闲状态，IDLE 也只置位一次。IDLE=0，没有检测到空闲线路；IDLE=1，检测到空闲线路。

D3（OR）—接收器溢出标记位。当一个新的字符准备转移到接收数据寄存器（缓冲器），但以前接收到的字符还未从 UARTx_D 读取时，OR 置位。要清除 OR，应先读 UARTx_S1 中的 OR，然后读 UARTx_D。OR=0，没有溢出；OR=1，接收溢出（新 UART 数据丢失）。

D2（NF）—噪声标志。在 UART 接收器中采用了高级采样技术，对每一个接收到的位进行 3 次采样。如果任意一次采样与其他采样不同，将置位 NF。NF=0，未检测到噪声；NF=1，在 UARTx_D 的接收数据中检测到噪声。

D1（FE）—帧错误标志。如果在应该出现停止位的时刻，检测到 0，则该位置位。要清除 FE，可先读 UARTx_S1，再读 UARTx_D。FE=0，未检测到成帧错误，这不能保证成帧正确；FE=1，成帧错误。

D0（PF）—奇偶校验错误标志。当奇偶校验使能（PE=1），且接收到数据的奇偶校验位于期望的奇偶校验值不匹配，该位置位。PF=0，没有奇偶效验错误；PF=1，奇偶效验错误。

4．波特率寄存器：UARTx_BDH、UARTx_BDL

UARTx_BDH 寄存器与 UARTx_BDL 寄存器一同控制着波特率生成器的分频因子。只有当收发器都禁用的情况下，13 位[SBR12：SBR0]波特率设置位才可被设置。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	LBKDIE	RXEDGIE	SBNS	SBR				
复位	0							

D7（LBKDIE）—LIN 中止检测中断使能。LBKDIE=0，UART_S2[LBKDIF]会禁止硬件中断（通过轮询机制）。LBKDIE=1，当 UART_S2[LBKDIF]标志为 1 时，有硬件中断请求。

D6（RXEDGIE）—RX 输入有效边沿中断允许位。RXEDGIE=0，UART_S2[RXEDGIF]会禁止硬件中断（通过轮询机制）。RXEDGIE=1，当 UART_S2[RXEDGIF]标志为 1 时，有硬件中断请求。

D5（SBNS）—停止位数选择。SBNS 决定了停止位的位数。只有当收发器同时处于禁止情况下该位才可以被改变。SBNS=0，一位停止位；SBNS=1，两位停止位。

D4～D0（SBR）—波特率模数因子。它与波特率低字节寄存器(UARTx_BDL)的 8 位组合成 13 位[SBR12：SBR0]，统称为 BR，用于给波特率生成器设置模数因子值。

UARTx_BDL 与 UARTx_BDH 寄存器一起控制着 UART 波特率生成器的预分频因子。只有当收发器都禁用的情况下，13 位[SBR12:SBR0]波特率设置位（波特率模数因子）才可被设置。8 位 UARTx_BDL 寄存器为波特率设置位的低 8 位[SBR7:SBR0]，复位后 UART_BDL 为非零值，复位后波特率生成器被禁止。

SBR[12:0]中的 13 位统称为 BR。它们给波特率生成器设置模数因子值，当 BR 是 1～8191 时，UART 波特率=BUSCKL/（16×BR）。

5．数据寄存器

UARTx_D（*x*=0～2）其实是三个单独的 8 位寄存器，读时会返回只读接收数据寄存器中的内容，写时会写到只写发送数据寄存器。

6.3.3　UART 驱动构件的实现

UART 驱动构件存放于网上光盘“..\CH06-KEA128-UART\05_Driver\uart”文件夹，供复制使用，各个工程文件夹下的“..\05_Driver\uart”文件夹中 UART 驱动构件与此一致。

1．UART 驱动构件的源程序文件（uart.c）

UART 驱动构件的源程序文件中实现的对外接口函数，主要是对相关寄存器进行配置，从而完成构件的基本功能。

```
//============================================================================
//文件名称：uart.c
//功能概要：uart 底层驱动构件源文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-06-05  V2.0
//芯片类型：KEA128
//============================================================================
#include "uart.h"

//=====串口 0、1、2 号地址映射====
const UART_MemMapPtr UART_ARR[] = {UART0_BASE_PTR,
                                   UART1_BASE_PTR,
                                   UART2_BASE_PTR};
//====定义串口 IRQ 号对应表====
static uint_8 table_irq_uart[3] = {12, 13, 14};

//============================================================================
//函数名称：uart_init
//功能概要：使用总线时钟（BUS_CLK_KHZ），初始化 uart 模块
//参数说明：uartNo:串口号 UART_0、UART_1、UART_2
//         baud:波特率 1200、2400、4800、9600、19200、115200
//函数返回：无
//============================================================================
 void  uart_init (uint_8 uartNo,uint_32 baud_rate)
```

```
{
    //局部变量声明
    register uint_16 sbr;
    uint_8 temp;
    //声明 uartch1 为 UART_MemMapPtr 结构体类型指针
    UART_MemMapPtr uartch1 = UART_ARR[uartNo];

    //根据带入参数 uartNo，给局部变量 uartch1 赋值
    switch (uartNo)
    {
    case UART_0:
        #if (UART_0_GROUP==1)
            //复用引脚 PTA2 和 PTA3 分别为 UART0 的接收和发送
            SIM_PINSEL0 |= SIM_PINSEL_UART0PS_MASK;
        #endif

        #if (UART_0_GROUP==2)
            //复用引脚 PTB0 和 PTB1 分别为 UART0 的接收和发送
            SIM_PINSEL0 &= ～SIM_PINSEL_UART0PS_MASK;
        #endif
        SIM_SCGC |= SIM_SCGC_UART0_MASK;        //UART0 使能总线时钟
        UART0_C2 |= UART_C2_TE_MASK;            //使能 UART0 发送功能
        UART0_C2 |= UART_C2_RE_MASK;            //使能 UART0 接收功能
        UART0_C2 |= UART_C2_RIE_MASK;           //使能 UART0 接收中断功能
    case UART_1:
        //复用引脚 PTC6 和 PTC7 分别为 UART1 的接收和发送
        #if (UART_1_GROUP==1)
            SIM_PINSEL1 &= ～SIM_PINSEL1_UART1PS_MASK;
        #endif
        //复用引脚 PTF2 和 PTF3 分别为 UART1 的接收和发送
        #if (UART_1_GROUP==2)
            SIM_PINSEL1 |= SIM_PINSEL1_UART1PS_MASK;
        #endif

        SIM_SCGC |= SIM_SCGC_UART1_MASK;        //UART1 使能总线时钟
        UART1_C2 |= UART_C2_TE_MASK;            //使能 UART1 发送功能
        UART1_C2 |= UART_C2_RE_MASK;            //使能 UART1 接收功能
        UART1_C2 |= UART_C2_RIE_MASK;           //使能 UART1 接收中断功能

        break;
    case UART_2:
        //复用引脚 PTD6 和 PTD7 分别为 UART2 的接收和发送
        #if (UART_2_GROUP==1)
        SIM_PINSEL1  &= ～ SIM_PINSEL1_UART2PS_MASK;
        #endif
        #if (UART_2_GROUP==2)
            //复用引脚 PTI0 和 PTI1 分别为 UART2 的接收和发送
```

```
            SIM_PINSEL1  |= SIM_PINSEL1_UART2PS_MASK;
        #endif

        SIM_SCGC |= SIM_SCGC_UART2_MASK;        //UART2 使能总线时钟
        UART2_C2 |= UART_C2_TE_MASK;            //使能 UART2 发送功能
        UART2_C2 |= UART_C2_RE_MASK;            //使能 UART2 接收功能
        UART2_C2 |= UART_C2_RIE_MASK;           //使能 UART2 接收中断功能
        break;

    default:
        break;                                  //传参错误，返回
    }

     //暂时关闭串口发送与接收功能
    uartch1->C2 &= ～(UART_C2_TE_MASK | UART_C2_RE_MASK);

    //配置波特率使用 Bus clock = 24 MHz 总线时钟
    //配置串口工作模式,8 位无校验模式
    uartch1->C1 = 0;
    sbr = (uint_16) ((BUS_CLK_KHZ * 1000) / (baud_rate * 16));
    temp = UART_BDH_REG(uartch1) & ～(UART_BDH_SBR(0x1F));
    UART_BDH_REG(uartch1) = temp | UART_BDH_SBR(((sbr & 0x1F00) >> 8));
    UART_BDL_REG(uartch1) = (uint_8) (sbr & UART_BDL_SBR_MASK);

    //初始化控制寄存器、清标志位
    uartch1->C1 = 0x00;
    uartch1->C3 = 0x00;
    uartch1->S2 = 0x00;
    //启动发送接收
    uartch1->C2 |= (UART_C2_TE_MASK | UART_C2_RE_MASK);
}

//=============================================================================
 //函数名称：uart_send1
 //参数说明：uartNo:串口号 UART_0、UART_1、UART_2
 //         data:要发送的字节
 //函数返回：函数执行状态，0=正常，非 0=异常。
 //功能概要：串行发送 1 个字节

//=============================================================================
 uint_8 uart_send1(uint_8 uartNo, uint_8 data)
{
  uint_8 t;
  //声明 uartch1 为 UART_MemMapPtr 结构体类型指针
  UART_MemMapPtr uartch1 = UART_ARR[uartNo];

  for (t = 0; t < 0xFBBB; t++)                  //查询指定次数
```

```
        {
            if ((uartch1->S1) & UART_S1_TDRE_MASK)  //判断发送缓冲区是否为空
            {
                uartch1->D = data;                  //获取数据并发送
                break;                              //跳出循环
            }

        } //end for
        if (t >= 0xFBBB)
            return 1;                               //发送超时，发送失败
        else
            return 0;                               //成功发送
    }

//========================================================================
     //函数名称：uart_sendN
     //参数说明：uartNo：串口号 UART_0、UART_1、UART_2
     //          buff：发送缓冲区
     //          len:发送字节数
     //函数返回：函数执行状态，0=正常，1=异常
     //功能概要：串行发送多个字节

//========================================================================
     uint_8 uart_sendN (uint_8 uartNo ,uint_16 len ,uint_8* buff)
    {
        uint_16 i;
        for (i = 0; i < len; i++)
        {
            if (uart_send1(uartNo, buff[i]))    //发送一个字节数据，失败则跳出循环
            {
                break;
            }
        }
        if (i >= len)
            return 1;   //发送 len 个字节出错
        else
            return 0;   //发送 len 个字节成功
    }

    //========================================================================
    //函数名称：uart_send_string
    //参数说明：uartNo:UART 模块号 UART_0、UART_1、UART_2
    //          buff:要发送的字符串的首地址
    //函数返回：函数执行状态，0=正常，非 0=异常
    //功能概要：从指定 UART 端口发送一个以'\0'结束的字符串(以双引号表达的字符串，即以'\0'
    //          结束的字符串)
    //例：uart_send_string(UART_0,"abcdefg"); 即可发送字符  abcdefg
```

```
//========================================================================
uint_8 uart_send_string(uint_8 uartNo, uint_8 *buff)
{
    uint_16 i;
    for(i = 0; *(buff+i) != '\0'; i++)          //遍历字符串里的字符
    {
        if (uart_send1(uartNo,*(buff+i)))       //发送指针对应的字符
            return 1;  //发送失败
    }
    return 0;         //发送成功
}

//========================================================================
//函数名称：uart_re1
//参数说明：uartNo：串口号:UART_0、UART_1、UART_2
//         fp:接收成功标志的指针，*fp=0，成功接收；*fp=1，接收失败
//函数返回：接收返回字节
//功能概要：串行接收 1 个字节
//========================================================================
uint_8 uart_re1 (uint_8 uartNo,uint_8 *fp)
{
    uint_32 t;
    uint_8  dat;
    UART_MemMapPtr uartch1 = UART_ARR[uartNo]; //获取 UART1 或者 2 基地址

    for (t = 0; t < 0xFBBB; t++)//查询指定次数
    {
        //判断接收缓冲区是否满
        if ((uartch1->S1) & UART_S1_RDRF_MASK)
        {
            dat = uartch1->D;                   //获取数据
            *fp = 0;                            //接收成功
            break;
        }
    }//end for
    if(t >= 0xFBBB)
    {
        dat = 0xFF;
        *fp = 1;                                //未收到数据
    }
    return dat;                                 //返回接收到的数据

}
//========================================================================
//函数名称：uart_reN
//参数说明：uartNo：串口号 UART_0、UART_1、UART_2
//         buff：接收缓冲区
```

```
//          len:接收长度
//函数返回：函数执行状态，0=正常，非 0=异常
//功能概要：串行接收 n 个字节
//=========================================================================
uint_8 uart_reN (uint_8 uartNo ,uint_16 len ,uint_8* buff)
{
    uint_16 i;
    uint_8 flag = 0;

    //判断是否能接收数据
    for (i = 0; i < len && 0 == flag; i++)
    {
        buff[i] = uart_re1(uartNo, &flag);      //接收数据
    }
    if (i >= len)
        return 1;                               //接收失败
    else
        return 0;                               //接收成功
}

//=========================================================================
//函数名称：uart_enable_re_int
//参数说明：uartNo：串口号 UART_0、UART_1、UART_2
//函数返回：无
//功能概要：开串口接收中断
//=========================================================================
void uart_enable_re_int(uint_8 uartNo)
{
    UART_MemMapPtr uartch1 = UART_ARR[uartNo];

    uartch1->C2 |= UART_C2_RIE_MASK;            //开放 UART 接收中断
    NVIC_EnableIRQ(table_irq_uart[uartNo]);     //开中断控制器 IRQ 中断
}

//=========================================================================
//函数名称：uart_disable_re_int
//参数说明：uartNo：串口号 UART_0、UART_1、UART_2
//函数返回：无
//功能概要：关串口接收中断
//=========================================================================
void uart_disable_re_int(uint_8 uartNo)
{
    UART_MemMapPtr uartch1 = UART_ARR[uartNo];
    uartch1->C2 &= ~UART_C2_RIE_MASK;           //禁止 UART 接收中断
    NVIC_DisableIRQ(table_irq_uart[uartNo]);    //禁止中断控制器 IRQ 中断
}
```

第 7 章

Systick、RTC、PWT 及 PIT

本章导读：本章阐述 KEA128 与时钟相关的几个模块的编程，主要内容有：7.1 节给出内核时钟 SysTick；7.2 节给出实时时钟 RTC 模块；7.3 节给出脉冲宽度定时器 PWT；7.4 节给出周期性中断定时器 PIT。每一节中，先简单介绍模块基本功能及其编程要点，然后通过对寄存器分析来给出相应的编程指导。

本章参考资料：本章 7.1 节（SysTick）参考自《M0+参考手册》；7.2～7.4 节分别参考自《KEA128 参考手册》的第 29 章（RTC）、第 27 章（PWT）及第 28 章（PIT）。

7.1 ARM Cortex-M0+内核时钟（Systick）

ARM Cortex-M 内核中包含了一个简单的定时器 SysTick，又称为滴答定时器。SysTick 定时器被捆绑在 NVIC 中，有效位数是 24 位，采用减 1 计数的方式工作，当减 1 计数到 0，可产生 SysTick 异常（中断），中断号为 15。

嵌入式操作系统或使用了时基的嵌入式应用系统，都必须由一个硬件定时器来产生需要的滴答中断，作为整个系统的时基。由于所有使用 Cortex-M 内核的芯片都带有 SysTick，并且在这些芯片中，SysTick 的处理方式（寄存器映射地址及作用）都是相同的，若使用 SysTick 产生时间滴答，可以简化嵌入式软件在 Cortex-M 内核芯片间的移植工作。

7.1.1 Systick 模块的编程结构

1．Systick 定时器模块的寄存器地址

SysTick 定时器模块中有 4 个 32 位寄存器，其映像地址及简明功能见表 7-1。

表 7-1 SysTick 模块的寄存器映像地址及简明功能

寄存器名	简　称	访问地址	简明功能
控制及状态寄存器	SYST_CSR	0xE000_E010	配置功能及状态标志
重载寄存器	SYST_RVR	0xE000_E014	低 24 位有效，计数器到 0，用该寄存器的值重载
计数器	SYST_CVR	0xE000_E018	低 24 位有效，计数器当前值，减 1 计数
校准寄存器	SYST_CALIB	0xE000_E01C	针对不同 MCU，校准恒定中断频率。KEA 未用

2．控制及状态寄存器 SYST_CSR

控制及状态寄存器 SYST_CSR 见表 7-2。主要有溢出标志位 COUNTFLAG、时钟源选择位 CLKSOURCE、中断使能控制位 TICKINT 和 SysTick 模块使能位 ENABLE。复位时，各位为 0。

表 7-2 控制及状态寄存器 SYST_CSR

位	名 称	R/W	功 能 说 明
16	COUNTFLAG	R	计数器减 1 计数到 0，则该位为 1；读取该位清 0
2	CLKSOURCE	R/W	=0，内部时钟/16；=1，内核时钟（KEA128 无外部时钟，该位读写始终为 1）①
1	TICKINT	R/W	=0，禁止中断；=1，允许中断（计数器到 0 时，中断）
0	ENABLE	R/W	SysTick 模块使能位，=0，关闭；=1，使能

3．计数器 SYST_CVR 及重载寄存器 SYST_RVR

SysTick 模块的计数器 SYST_CVR 保存当前计数值，这个寄存器是由芯片硬件自行维护，用户无须干预，系统可通过读取该寄存器的值得到更精细的时间表示。SysTick 模块的重载寄存器 SYST_RVR 的低 24 位 D23～D0（RELOAD）有效，其值是计数器的初值及重载值。

SysTick 模块内的计数器 SYST_CVR 是一个 24 位计数器，减 1 计数。初始化时，选择时钟源（决定了计数频率）、设置重载寄存器 SYST_RVR、设置优先级、允许中断，计数器的初值为重载寄存器 SYST_RVR 中的值、使能该模块。则计数器开始减 1 计数，计数到 0 时，SysTick 控制及状态寄存器 SYST_CSR 的溢出标志位 COUNTFLAG 被置 1，产生中断请求，同时，计数器自动重载初值并继续减 1 计数。

4．Cortex-M0+内核优先级设置寄存器

编写 SysTick 模块的初始化程序还需用到 Cortex-M0+内核优先级设置寄存器（System Handler Priority Register 3，SHPR3），用于设定 SysTick 模块中断的优先级，位于系统控制块 SCB 中。在 ARM Cortex-M0+中，只有 SysTick、SVC（Supervisor Call，系统服务调用）和 PendSV（可挂起系统调用）等内部异常可以设置其中断优先级，其他内核异常的优先级是固定的。SVC 的优先级在 SHPR2 寄存器中设置，SysTick 和 PendSV 优先级在 SHPR3 寄存器中设置，见图 7-1。

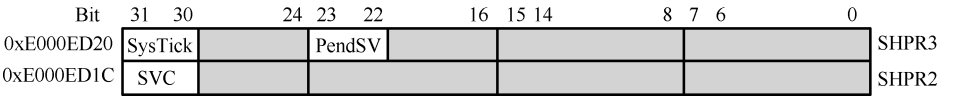


图 7-1 Systick 优先级寄存器

SHPR3 寄存器的最高两位 D31～D30 用于设置 SysTick 优先级，优先级可以设为 0～3 级，一般设为 2。

7.1.2 Systick 构件设计及测试实例

下面以 Systick 定时器模块为时钟源，每隔 1 s 钟通过串口向 PC 发送时钟、分钟和秒钟的应用。

① 参考《ARMv6-M Architecture Reference Manual》B3-278 页

1．Systick 构件设计

1）Systick 构件头文件（systick.h）

```
//=============================================================================
//文件名称：systick.h
//功能概要：systick 定时器模块构件头文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2015-05-08   V1.0
//=============================================================================

#ifndef _SYSTICK_H
#define _SYSTICK_H

#include "common.h"


//时钟源宏定义
#define CORE_CLOCK       1
#define CORE_CLOCK_DIV_16 0
//=============================================================================
//函数名称：systick_init
//函数返回：无
//参数说明：clk_src_sel：时钟源选择，1 表示内核时钟（core_clk_khz）；
//                                0 表示内核时钟/16
//         int_ms:中断的时间间隔。单位 ms 推荐选用 5,10,...,最大为 50
//功能概要：初始化 SysTick 模块，设置中断的时间间隔
//说    明：内核时钟频率 core_clk_khz 宏定义在 sysinit.h 中
//=============================================================================
void systick_init(uint_8 clk_src_sel, uint_8 int_ms);

#endif
```

2）Systick 构件源文件（systick.c）

```
//=============================================================================
//文件名称：systick.c
//功能概要：systick 定时器模块构件源文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2015-05-08   V1.0
//=============================================================================
#include "systick.h"

//=============================================================================
//函数名称：systick_init
//函数返回：无
//参数说明：clk_src_sel：时钟源选择，1 表示内核时钟（CORE_CLK_KHZ）；
//                                0 表示内核时钟/16
//         int_ms:中断的时间间隔，单位 ms，推荐选用 5,10,...,最大为 50
```

```
//功能概要：初始化 SysTick 模块，设置中断的时间间隔
//说    明：内核时钟频率 core_clk_khz 宏定义在 sysinit.h 中
//=========================================================================
void systick_init(uint_8 clk_src_sel, uint_8 int_ms)
{
    SysTick->CTRL = 0;                  //设置前先关闭 systick
    SysTick->VAL  = 0;                  //清除计数器
    //根据计数频率，确定并设置重载寄存器的值
    if(0 == clk_src_sel)                //0:内核时钟/16
    {
        SysTick->LOAD = CORE_CLK_KHZ*int_ms/16;
     }
    else                                //1:内核时钟
    {
        SysTick->LOAD = CORE_CLK_KHZ*int_ms;
        clk_src_sel=1;                  //防止入口参数错，提高程序健壮性
    }
    //设定 SysTick 优先级为 2
    SCB->SHP[2] |=(0x02)<<30;

    //设置时钟源,允许中断,使能该模块,开始计数
    SysTick->CTRL = (clk_src_sel<<SysTick_CTRL_CLKSOURCE_Pos) |
                    SysTick_CTRL_ENABLE_Msk |
                    SysTick_CTRL_TICKINT_Msk;
}
```

2．Systick 构件测试工程

测试工程的功能概述如下。

（1）串口通信格式：波特率 9600，1 位停止位，无校验。

（2）Systick 使用内核时钟 48 MHz。

（3）上电或按复位按钮时，调试串口输出“Hello SysTick”。

（4）Systick 每 10 ms 中断一次，在中断里再进行计数判断，每 100 个 Systick 中断 LIGHT_0～LIGHT_3 小灯状态改变(LIGHT_0～LIGHT_3 初始化时 2 亮 2 暗)，同时调试串口输出“systick isr”。

（5）PC 向 MCU 发送数据时，MCU 进入串口接收中断，将接收的一个字节直接回发。三个串口具有相同功能。

测试工程位于网上光盘中的“..\program\CH07-KEA128-Systick”文件夹。

7.2 实时时钟模块（RTC）

7.2.1 RTC 模块概述与编程要点

1．RTC 编程知识要素

实时计数器(RTC)能够用于时钟、日历或者任何其他与时间相关的任务，也可以用于周期性从低功耗模式唤醒，或在没有外部需求的情况下停止或者等待。

KEA128 的 RTC 的有效位数为 16 位，采用加 1 计数。在模块初始化后，每个时钟的上升沿，RTC_CNT 寄存器的值会加 1，直至与 RTC_MOD 寄存器中值相同时，置 RTC_SC[RFIF]位，并进入对应中断服务例程。在中断向量表中，RTC 的 IRQ 号为 20。

RTC 模块采用的时钟可以通过 RTC_SC[RTCLKS]字段选择，00 选择外部时钟源，01 选择实时时钟源是 1kHz（LPOCLK），10 选择内部参考时钟（ICSIRCLK），11 选择总线时钟。

同时可以通过 RTC_SC[RTCPS]字段配合选择预分频值。选定时钟和预分频后的周期可以参见表 7-3，此表中的时钟只是举例，实际时钟应该配置的时钟为准。

表 7-3　不同时钟下预分频周期

RTCPS	32768 Hz OSC 时钟源预分频周期（RTCLKS=00）	LPO 时钟源预分频期（RTCLKS=01）	32768 Hz 内部参考时钟源预分频周期（RTCLKS=10）	8 MHz 总线时钟源预分频周期（RTCLKS=11）
000	关闭	关闭	关闭	关闭
001	30.5176 μs	128 ms	30.5176 μs	16 μs
010	61.0351 μs	256 ms	61.0351 μs	32 μs
011	122.0703 μs	512 ms	122.0703 μs	64 μs
100	244.1406 μs	1024 ms	244.1406 μs	128 μs
101	488.28125 μs	2048 ms	488.28125 μs	256 μs
110	976.5625 μs	100 ms	976.5625 μs	12.5 μs
111	1.9531 μs	1 s	1.9531 μs	125 μs

KEA128 的 RTC 模块有一个输出的引脚用作 RTCO，提供给外部构件使用。在 RTC 产生溢出时，会反转此此引脚的输出，本板上使用的是 PTC5 引脚。可以通过设置 RTC_SC[RTCO]位来设置是否启用输出引脚。

2．寄存器地址分析

表 7-4 提供了一个在 PIT 模块中的所有可访问的寄存器的详细信息。

表 7-4　RTC 存储器映射

绝 对 地 址	寄存器名称	宽度（位）	操　作	置 位 值
4003_D000	RTC 状态和控制寄存器（RTC_SC）	32	R/W	0000_0000h
4003_D004	RTC 模数寄存器（RTC_MOD）	32	R/W	0000_0000h
4003_D008	RTC 计数寄存器（RTC_CNT）	32	R	0000_0000h

3．控制寄存器

RTC_SC 包含实时中断状态标志（RTIF）和触发器输出使能位（RTCO）。

高 16 位为保留位																
数据位	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
读	RTCLKS		0			RTCPS			RFIF	RFIE	0	RTCO	0			
写																
复位	0															

下面给出相关的功能说明，未说明的位或是字段均为保留字段（只读，值为 0）。

D15～D14（RTCLKS）—实时时钟源选择。该读/写字段选择时钟源输入到 RTC 时钟源,改变时钟源便可清预分频器和 RTC_CNT 计数器、重置 RTCLKS 为 0。00 选择外部时钟源，01 选择实时时钟源是 1kHz（LPOCLK），10 选择内部参考时钟（ICSIRCLK），11 选择总线时钟。

D10～D8（RTCPS）—实时时钟分频器选择。该读/写字段选择时钟源基于二进制或十进制的分频值，具体如下。

000：关。

001：如果 RTCLKS=x0，分频值为 1；如果 RTCLKS=x1，分频值为 128。

010：如果 RTCLKS=x0，分频值为 2；如果 RTCLKS=x1，分频值为 256。

011：如果 RTCLKS=x0，分频值为 4；如果 RTCLKS=x1，分频值为 512。

100：如果 RTCLKS=x0，分频值为 8；如果 RTCLKS=x1，分频值为 1024。

101：如果 RTCLKS=x0，分频值为 16；如果 RTCLKS=x1，分频值为 2048。

110：如果 RTCLKS=x0，分频值为 32；如果 RTCLKS=x1，分频值为 100。

111：如果 RTCLKS=x0，分频值为 64；如果 RTCLKS=x1，分频值为 1000。

D7（RFIF）—实时中断标记。此状态位表示 RTC_CNT 计数器寄存器达到在 RTC_MOD 模数寄存器的值，写逻辑 0 没有作用，写入逻辑 1 该位清 0，发生实时中断请求。复位后 RTIF 清 0。0 表示 RTC 计数器没有达到 RTC 模数寄存器的值，1 表示 RTC 计数器达到 RTC 模数寄存器的值。

D6（RFIE）—实时中断使能。使能实时中断，0 表示实时中断请求禁止，1 表示实时中断请求使能。

D4（RTCO）—实时计数器输出。使能实时输出到输出引脚，当 RTC 计数器溢出，如果该位设置使能，RTCO 的输出引脚将反转。0 表示实时计数器输出禁止，1 表示实时计数器输出使能。

4．其他寄存器

1）RTC 模数寄存器（RTC_MOD）

D15～D0（MOD）—RTC 模数。该字段是 16 位数值，可以设置为 0x0000～0xFFFF 之间的值。当 RTC 计数寄存器（RTC_CNT）的值到达 RTC 模数寄存器（RTC_MOD）的值时，置位 SC[RTIF]状态字段，RTC_CNT 复位到 0x0000。复位后模数值为 0x0000。

2）RTC 计数寄存器（RTC_CNT）

D31～D0（CNT）—RTC 计数器。增 1 计数器，高 16 位保留，该只读字段包含 16 位计数器的当前值，首先读 CNT[7:0]，然后读[15:8]。写操作对该寄存器没有影响。复位或者写不同的值到 SC[RTCLKS]和 SC[RTCPS]会清除计数器使其值为 0x0000。

7.2.2 RTC 构件设计及测试实例

本程序的功能是每隔一定时间产生一次中断，输出相应信息。RTC 具有初始化、启动和关闭等基本操作，以及 RTC 中断处理函数。按照构件的思想，可将它们封装成独立的功能函

数，初始化完成对 RTC 模块的工作属性的设定。对 RTC 进行编程，实际上已经涉及对硬件底层寄存器的直接操作，因此可以将初始化、启动和关闭等基本操作所对应的功能函数共同定义在名为 rtc.c 的文件中，并按照相对严格的构件设计原则对其进行封装，同时配以命名为 rtc.h 的头文件，用来定义模块的基本信息和对外接口。将中断函数定义在名为 isr.c 的文件中。

下面通过具体 RTC 基本功能函数来进一步深刻理解 RTC 构件化编程思想

1．RTC 构件头文件

```
//============================================================================
//文件名称：rtc.h
//功能概要：rtc 底层驱动构件头文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-5-8  V1.0
//============================================================================
#ifndef RTC_H
#define RTC_H

#include "common.h"

#define   RTC_IRQ_NO    20

#define   RTC_LPOCLK    1
#define   RTC_BUSCLK    0

//============================================================================
//函数名称：rtc_init
//函数参数：sel_clk:rtc 模块的时钟源选择，1=1kHz；0=BUSCLK
//         int_ms:中断的秒数,sel_clk=1 时，int_ms 可以从 1 到 3276800；
//                          sel_clk=0 时，int_ms 可以从 1 到 1365
//函数返回：无
//功能概要：RTC 驱动初始化
//============================================================================
void rtc_init(uint_16 sel_clk,uint_16 int_ms);

#endif
```

2．RTC 构件的源程序文件

```
//============================================================================
//文件名称：rtc.h
//功能概要：rtc 底层驱动构件头文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-5-8  V1.0
//============================================================================

#include "rtc.h"
```

```
//==========================================================================
//函数名称：rtc_init
//函数参数：sel_clk:rtc 模块的时钟源选择，1=1kHz；0=BUSCLK
//         int_ms:中断的秒数,sel_clk=1 时，int_ms 可以从 1 到 3276800；
//                          sel_clk=0 时，int_ms 可以从 1 到 1365
//函数返回：无
//功能概要：RTC 驱动初始化
//==========================================================================
void rtc_init(uint_16 sel_clk,uint_16 int_ms)
{
    uint_16 mod_value;
    SIM_SCGC |= 0x1u;    //开时钟门

    NVIC_DisableIRQ(RTC_IRQ_NO);        //禁用 RTC 中断

    //配置 RTC 模块
    if(1 == sel_clk)
    {
RTC_SC |= RTC_SC_RTCLKS(0b01);          //选择时钟源；1 kHz
RTC_SC |= RTC_SC_RTCPS(0b110);;         //选择预分频:100
RTC_SC |= RTC_SC_RTIF_MASK | RTC_SC_RTIE_MASK;//中断清除使能

        mod_value = int_ms/100;       //mod_value = int_ms * (1000/100)/1000;
    }
    else
    {
        RTC_SC |= RTC_SC_RTCLKS(0b11);  //选择时钟源；总线时钟 24 MHz
        RTC_SC |= RTC_SC_RTCPS(0b111);  //选择预分频:1000
        RTC_SC |= RTC_SC_RTIF_MASK | RTC_SC_RTIE_MASK; //中断清除使能
        RTC_SC |= RTC_SC_RTCO_MASK;                    //使能输出
        SIM_PINSEL0 |= SIM_PINSEL_RTCPS_MASK;          //输出选择 PTC5

        mod_value = int_ms * 24;    //mod_value = int_ms * BUS_CLK_KHZ/1000
    }

    RTC_MOD = mod_value;            //设置 MOD 值

    NVIC_EnableIRQ(RTC_IRQ_NO);     //使能 RTC 中断
}
```

RTC 中断处理函数

```
//rtc 中断服务例程
void isr_rtc(void)
{
    RTC_SC |= RTC_SC_RTIF_MASK;

light_change(LIGHT_0);              //灯 0（LIGHT_0）状态变化
light_change(LIGHT_1);              //灯 1（LIGHT_1）状态变化
```

```
    light_change(LIGHT_2);              //灯 2（LIGHT_2）状态变化
    light_change(LIGHT_3);              //灯 3（LIGHT_3）状态变化
       printf("rtc isr\r\n");
    }
```

3．RTC 测试工程

测试工程功能概述如下。

（1）串口通信格式：波特率 9600，1 位停止位，无校验。

（2）RTC 使用内部 1 kHz 时钟。

（3）上电或按复位按钮时，调试串口输出“Hello RTC!”。

（4）RTC 每 1 s 中断一次，在中断里 LIGHT_0～LIGHT_3 小灯状态改变（LIGHT_0～LIGHT_3 初始化时 2 亮 2 暗），同时调试串口输出“rtc isr”。

（5）PC 向 MCU 发送数据时，MCU 进入串口接收中断，将接收的一个字节直接回发。三个串口具有相同功能。

PWT 构件的测试工程位于网上光盘中的“..\program\CH07-KEA128-RTC”文件夹。

7.3　脉冲宽度定时器（PWT）

7.3.1　PWT 模块概述与编程要点

1．PWT 模块编程知识要素

PWT 模块实现了输入捕捉的功能，可以用于计算输入波形的频率及占空比等。

PWT 模块的时钟通过 PWT_R1[PCLKS]位选择，PCLKS=0 时，BUS_CLK 为 PWT 计数器的时钟源；PCLKS=1 时，ALTCLK 为 PWT 计数器的时钟源。同时可以配置 PWT_R1[PRE]字段可以对时钟预分频，PRE=000，时钟/1；PRE=001，时钟/2；PRE=010，时钟/4；PRE=011，时钟/8；PRE=100，时钟/16；PRE=101，时钟/32；PRE=110，时钟/64；PRE=111，时钟/128。

PWT 所用引脚，可以参看表 7-5。

表 7-5　PWT 信号描述

信　　号	I/O	描　　述
PWTIN[3:0]	输入	脉冲宽度定时器输入捕捉
ALTCLK	输入	外部时钟输入引脚

PWT 的边沿捕捉方式通过 PWT_R1[EDGE]配置，下面以 EDGE=01 为例来说明 PWT 的工作方式。当 EDGE=01 时，PWT 计数器会在遇到输入的第一个上升沿开始计数，遇到第一个下降沿时，将计数器的当前值写入到 PWT_R2[NPW]字段，在这过程中计数器依然在加 1 计数，当遇到第二个上升沿时，将计数器的当前值写入到 PWT_R1[PPW]字段，同时计数器置 0，此时通过读取 PWT_R1[PPW]和 PWT_R2[NPW]字段，可以计算输入脉冲占空比，并且获得时钟频率后，还可以计算得到输入脉冲的频率，从而实现对输入脉冲的捕捉分析。

2．寄存器地址分析

表 7-6 提供了 PWT 模块中的所有可访问的寄存器的详细信息。

表 7-6 PWT 内存映射

绝对地址	寄存器名	宽度	访问	复位
4003_3000	脉冲宽度定时器 1（PWT_R1）	32	R/W	0000_0000h
4003_3004	脉冲宽度定时器 2（PWT_R2）	32	R	0000_0000h

3．PWT 寄存器 1（PWT_R1）

该寄存器定义了 PWT 的通用控制和状态位，它还包含了正脉冲宽度的内容。

数据位	D31	D30	D29	D28	D27	D26	D25	D24	D23	D22	D21	D20	D19	D18	D17	D16
读	PPW															
写	—															
复位	0															

数据位	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
读	PCLKS	PINSEL		EDGE		PRE			PWTEN	PWTIE	PRDYIE	POVIE	0	0	PWTRDY	PWTOV
写													PWTSR	—		
复位	0															

下面给出相关的功能说明，未说明的位或是字段均为保留字段（只读，值为 0）。

D31～D16（PPW）—正脉冲宽度。捕捉的正脉冲宽度值，建议用半字（16 位）或字（32 位）读取正脉冲宽度值。

D15（PCLKS）—PWT 时钟源选择。控制 PWT 的时钟源选择，PCLKS=0 时，BUS_CLK 为 PWT 计数器的时钟源；PCLKS=1 时，ALTCLK 为 PWT 计数器的时钟源。

D14～D13（PINSEL）—PWT 脉冲输入引脚选择。如果 PWT 时钟源是外部时钟源，使能相应的 PWT 输入引脚。PINSEL=00，选择 PWTIN[0]引脚；PINSEL=01，选择 PWTIN[1]；PINSEL=10，选择 PWTIN[2]引脚；PINSEL=11，选择 PWTIN[3]引脚。

D12～D11（EDGE）—PWT 输入边沿灵敏度。用于选择哪种边沿触发测量脉冲宽度，以及哪种边缘触发捕捉信号。用户可以通过改变 EDGE[1:0]的值，改变触发和捕捉模式。改变 EDGE[1:0]后需要 PWT 软件复位。PWTEN 位先清 0，然后 PWTEN 位置 1，也同样具有复位的效果。EDGE=00，第一个下降沿处开始测量脉冲宽度，并在后续所有的下降沿捕捉脉冲宽度；EDGE=01，第一个上升沿处开始测量脉冲宽度，并在后续所有的上升和下降沿捕捉脉冲宽度；EDGE=10，第一个下降沿处开始测量脉冲宽度，并在后续所有的上升和下降沿捕捉脉冲宽度；EDGE=11，第一个上升沿处开始测量脉冲宽度，并在后续所有的上升沿捕捉脉冲宽度。

D10～D8（PRE）—PWT 时钟分频(CLKPRE)设置。设置 PWT 计数器的时钟分频值。PRE=000，时钟/1；PRE=001，时钟/2；PRE=010，时钟/4；PRE=011，时钟/8；PRE=100，时钟/16；PRE=101，时钟/32；PRE=110，时钟/64；PRE=111，时钟/128。

D7（PWTEN）—PWT 模块使能。使能/除能 PWT 模块，设置 PWTEN 后，不能改变 PWT 的配置，否则可能会出错。PWTEN=0，PWT 模块除能；PWTEN=1，PWT 模块使能。

D6（PWTIE）—PWT 模块中断使能。使能 PWT 模块产生中断，PWTIE=0，PWT 模块不能产生中断；PWTIE=1，PWT 模块能产生中断。

D5（PRDYIE）—PWT 脉冲宽度数据就绪中断使能。首先 PWTIE 位置 1，当 PWTRDY 置 1 时，使能/除能 PWT 模块产生中断。PRDYIE=0，当 PWTRDY 置 1，PWT 模块不能产生中断；PRDYIE=1，当 PWTRDY 置 1，PWT 模块能产生中断。

D4（POVIE）—PWT 计数器溢出中断使能。当 PWT 计数器溢出（PWTOV 置 1）时，使能/除能 PWT 模块产生中断。POVIE=0，当 PWTOV 置 1，PWT 模块不能产生中断；POVIE=1，当 PWTOV 置 1，PWT 模块能产生中断。

D3（PWTSR）—PWT 软复位。执行 PWT 软复位，读 PWTSR 位时一直为 0。PWTSR=0，不采取任何操作；PWTSR=1，向 PWTSR 写 1，PWT 将执行软复位。

D1（PWTRDY）—PWT 脉冲宽度有效。表明 PWT 脉宽寄存器已更新，并已准备好被读取。读 PWTRDY 位，PWTRDY 位会清 0，然后当 PWTRDY 置 1，向 PWTRDY 写 0。向 PWTRDY 写 1 是无用的。PWTRDY 位的设置与 EDGE[1:0]有关。PWTRDY=0，PWT 脉宽寄存器没有更新；PWTRDY=1，PWT 脉宽寄存器已更新。

D0（PWTOV）—PWT 计数器溢出。表明 PWT 计数器从 0xFFFF 运行到 0x0000。当 PWTOV 位置 1，向 PWTOV 位写 0，PWTOV 位会清 0。向 PWTOV 写 1 是无用的。如果产生了其他溢出，清 PWTOV 操作失败。PWTOV=0，PWT 计数器没有溢出；PWTOV=1，PWT 计数器从 0xFFFF 运行到 0x0000。

4．PWT 寄存器 2（PWT_R2）

脉冲宽度定时器 2 包含了负脉冲宽度的内容和 16 位自由运行计数器的内容。

PWT_R2 寄存器有 PWTC 和 NPW 两个字段，其中 D31～D16 为 PWT 计数器（PWTC），建议用半字（16 位）或字（32 位）读取 PWTC 值；D15～D0 负脉冲宽度（NPW），建议用半字（16 位）或字（32 位）读取负脉冲宽度值。

7.3.2　PWT 构件设计及测试实例

测试前，将 PTD5 和 PTC5 引脚相连接，使用 MicroUSB 口和 PC 相连接，在程序中 PWT 每次中断进行一次计时，并通过串口将计时信息发送给 PC。通过串口调试工具，可以看到脉冲测量的相关信息。PIT 模块具有初始化、PWT 中断禁用、PWT 中断使能及脉冲测量函数。按照构件的思想，可将它们封装成独立的功能函数。初始化完成对 PWT 模块的工作属性的设定。对 PWT 进行编程，实际上已经涉及对硬件底层寄存器的直接操作，因此可以将初始化、PWT 中断禁用、PWT 中断使能及脉冲测量函数等基本操作所对应的功能函数共同定义在命名为 PWT.c 的文件中，并按照相对严格的构件设计原则对其进行封装，同时配以命名为 PWT.h 的头文件，用来定义模块的基本信息和对外接口。将中断函数定义在名为 isr.c 的文件中。

下面通过封装 PWT 基本功能函数来进一步深刻理解 PWT 构件化编程思想。

1. PWT 构件头文件

```
//=============================================================================
//文件名称：PWT.h
//功能概要：PWT 底层驱动程序头文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-5-8  V2.0
//=============================================================================
#ifndef PWT_H
#define PWT_H

#include "includes.h"

#define PWT_IRQ_NUMBER    29
//=============================================================================
//函数名称：PWT_Init
//功能概要：PWT 初始化
//参数说明：无
//函数返回：无
//=============================================================================
void PWT_Init();
//=============================================================================
//函数名称：enable_pwt_int
//功能概要：PWT 中断使能
//参数说明：无
//函数返回：无
//=============================================================================
void enable_pwt_int();
//=============================================================================
//函数名称：disable_pwt_int
//功能概要：PWT 中断禁用
//参数说明：无
//函数返回：无
//=============================================================================
void disable_pwt_int();
//=============================================================================
//函数名称：Pulse_Measurement
//功能概要：脉冲测量
//参数说明：无
//函数返回：无
//=============================================================================
void Pulse_Measurement(void);

#endif
```

2. PWT 构件的源程序文件

```
//=============================================================================
//文件名称：PWT.c
//功能概要：PWT 底层驱动程序头文件
```

```
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-5-8  V2.0
//=============================================================================
#include "PWT.h"

//pwt_clock=clock source/prescaler, in this case 24MHz/128
const float pwt_clock=187500.0;

//=============================================================================
//函数名称：PWT_Init
//功能概要：PWT 初始化
//参数说明：无
//函数返回：无
//=============================================================================
void PWT_Init()
{
    SIM_SCGC |=  SIM_SCGC_PWT_MASK;
    PWT_R1|=PWT_R1_POVIE_MASK;              //使能溢出中断
    PWT_R1|=PWT_R1_PRDYIE_MASK;             //使能数据就绪中断
    PWT_R1|=PWT_R1_PWTIE_MASK;              //使能 PWT 模块中断
    PWT_R1|=PWT_R1_PWTEN_MASK;              //使能 PWT 模块
    PWT_R1|=PWT_R1_PRE(0b111);              //PWT 时钟源预分频选择 128
    PWT_R1|=PWT_R1_EDGE(0b01);              //从第一个上升沿可是测量
    PWT_R1|=PWT_R1_PINSEL(0b00);            //PWT 输入引脚选择 PTD5
}
//=============================================================================
//函数名称：enable_pwt_int
//功能概要：PWT 中断使能
//参数说明：无
//函数返回：无
//=============================================================================
void enable_pwt_int()
{
    PWT_R1|=PWT_R1_PWTEN_MASK;
    NVIC_EnableIRQ(PWT_IRQ_NUMBER);
}
//=============================================================================
//函数名称：disable_pwt_int
//功能概要：PWT 中断除能
//参数说明：无
//函数返回：无
//=============================================================================
void disable_pwt_int()
{
    NVIC_DisableIRQ(PWT_IRQ_NUMBER);
}
//=============================================================================
```

```
//函数名称：Pulse_Measurement
//功能概要：脉冲测量
//参数说明：无
//函数返回：无
//============================================================================
void Pulse_Measurement(void)
{
    frequency=(pwt_clock)/(positive_pulse + negative_pulse);//计算信号频临
    period = 1.0/frequency;                          //计算信号周期
    period=period*1000;                              //秒转化成毫秒

    decimals=frequency- (uint_8)frequency;
    decimals=decimals*10;
}
```

PWT 中断处理函数如下。

```
//PWT 中断处理函数
void  isr_pwt ()
{
    //如果发生了溢出中断
    if((PWT_R1&PWT_R1_PWTOV_MASK))
    {
         PWT_R1 ^= PWT_R1_PWTOV_MASK;                    //清溢出标识
         positive_pulse=0;
         negative_pulse=0;
         counter_overflow=1;
    }

    //如果发生了数据就绪中断
    if((PWT_R1&PWT_R1_PWTRDY_MASK))
    {
 (void)PWT_R1;                                           //读 PWT 寄存器 1
        counter_overflow=0;
        PWT_R1 ^= PWT_R1_PWTRDY_MASK;                    //清数据就绪标识
        positive_pulse=(PWT_R1>>16 & 0x0000FFFF);        //读正向测量值
        negative_pulse=PWT_R2 & 0x0000FFFF;              //读反向测量值
        (void)PWT_R1;                                    //读 PWT 寄存器 1
        PWT_R1 ^= PWT_R1_PWTRDY_MASK;                    //清数据就绪标识
    }
}
```

3．PWT 测试工程

测试工程功能概述如下。

（1）串口通信格式：波特率 9600，1 位停止位，无校验。

（2）PWT 使用定时器时钟 24 MHz。

（3）上电或按复位按钮时，调试串口输出“Hello PWT and RTC”。

（4）RTC 每 0.1 s 中断一次，在中断里 LIGHT_0～LIGHT_3 小灯状态改变（LIGHT_0～

LIGHT_3 初始化时 2 亮 2 暗）。

（5）测量脉冲宽度时（本程序是测量 RTC 模块产生的 5Hz 方波），串口输出：

```
"PWT data ready interrupt
Received: 4.9 Hz , Period: 200 ms
Positive Pulse: 18757, Negative Pulse: 18757"
```

（6）不测量脉冲宽度，即不连导线时，串口每 5 s 输出 5 行“PWT counter overflow”。

（7）PC 向 MCU 发送数据时，MCU 进入串口接收中断，将接收的一个字节直接回发。三个串口具有相同功能。

PWT 构件的测试工程位于网上光盘中的“..\program\CH07-KEA128-PWT”文件夹。

7.4　周期性中断定时器（PIT）

7.4.1　PIT 模块概述与编程要点

1．PIT 模块编程知识要素

周期性中断定时器模块（PIT）是一组可以用于产生中断和触发脉冲的定时器，该模块的中断都是可屏蔽的，每个定时器都有独立的计数器，此外，周期中断定时器模块没有外部引脚。

KEA128 的 PIT 模块有效位数是 32 位，采用减 1 计数。PIT 的时钟源不能选择，只能固定使用总线时钟，本书中将总线时钟设置为 24 MHz。

在 PIT 初始化使能后，会自动加载 PIT_LDVALn 寄存器值到 PIT_CVALn 寄存器，并开始减 1 计数，直到 0，产生中断，进入中断服务例程。在中断向量表中，RTC 的 IRQ 号为 22 + n，n 为选择的通道号，KEA128 中可以选择 0 或者 1。

除此之外，KEA128 的 PIT 模块还支持链接定时器，当一个定时器启用链接模式时，它只会在前一个计数器满时才开始计数，所以，如果定时器 n–1 倒计数至 0，那么计数器 n 将开始减量 1。这允许链接一些定时器到一起构成一个更长的定时器。

2．寄存器地址分析

表 7-7 提供了 PIT 模块中所有可访问的寄存器的详细信息。

表 7-7　PIT 模块中所有可访问的寄存器

绝 对 地 址	寄 存 器 名	宽　度	访　问	复　位
4003_7000	PIT 模块控制寄存器（PIT_MCR）	32	R/W	0000_0006h
4003_7100	定时器载入值寄存器（PIT_LDVAL0）	32	R/W	0000_0000h
4003_7104	当前定时值寄存器（PIT_CVAL0）	32	R	0000_0000h
4003_7108	定时器控制寄存器（PIT_TCTRL0）	32	R/W	0000_0000h
4003_710C	定时器标记寄存器（PIT_TFLG0）	32	R/W	0000_0000h
4003_7110	定时器载入值寄存器（PIT_LDVAL1）	32	R/W	0000_0000h
4003_7114	当前定时值寄存器（PIT_CVAL1）	32	R	0000_0000h

续表

绝对地址	寄存器名	宽度	访问	复位
4003_7118	定时器控制寄存器（PIT_TCTRL1）	32	R/W	0000_0000h
4003_711C	定时器标记寄存器（PIT_TFLG1）	32	R/W	0000_0000h

3．控制寄存器

1）PIT 模块控制寄存器（PIT_MCR）

高 16 位为保留位																
数据位	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
读	0													保留	MDIS	FRZ
写	—															
复位	0													1	1	0

下面给出相关的功能说明，未说明的位或是字段均为保留字段（只读，值为 0）。

D1（MDIS）—模块使能/禁用标准时钟。这个字段必须在任何其他设置完成之前被使能。MDIS=0，使能标准的 PIT 定时器时钟；MDI=1，禁用标准的 PIT 定时器时钟。

D0（FRZ）—冻结。当设备进入调试模式时，允许/停止定时器。FRZ=0，调试模式下继续运行；FRZ=1，定时器在调试模式下停止运行。

2）定时器控制寄存器（PIT_TCTRLn）

高 16 位为保留位																
数据位	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
读	0													CHN	TIE	TEN
写	—															
复位	0															

下面给出相关的功能说明，未说明的位或是字段均为保留字段（只读，值为 0）。

D2（CHN）—链接模式。当定时器 *n* 激活时，在定时器 *n* 减 1 前，定时器 *n*–1 必须是满值的。定时器 0 没有链接模式。CHN=0，定时器不被链接；CHN=1，定时器被链接到前一个定时器。例如，对于通道 2，如果字段 CHN 置 1，那么定时器 2 被链接到定时器 1。

D1（TIE）—定时器中断使能。如果使能中断，当中断被挂起或 TFLGn[TIF]位置 1 时，立即发生中断事件。为了避免这种情况，必须首先清相应的 TFLGn[TIF]位。TIE=0，禁止计时器 *n* 的中断请求；TIE=1，每当 TIF 位置 1，产生中断请求。

D0（TEN）—定时器使能。使能或禁用定时器。TEN=0，禁用定时器 *n*；TEN=1，使能定时器 *n*。

4．PIT 定时器中断标志寄存器(PIT_TFLGn)

32 位 PIT 定时器中断标志寄存器 PIT_TFLGn（*n*=0～1）中，D31～D1 均为保留位，D0 为定时器中断标志位（TIF）。TIF=1，发生超时，当 TIE=1 时，引发中断；TIF=0，未发生超时。TIF 为 w1c 位，对 TIF 位写 1 清 0。

5．其他寄存器

1）定时器载入值寄存器(PIT_LDVALn)

D31～D0 为定时器开始值（TSV），设置定时器开始值。定时器将向下计数到 0，然后它将生成一个中断并且再次载入这个寄存器值。写一个新值到这个寄存器将不会重启定时器，相反地，这个值将在定时器有效期内被装载。要想终止当前的循环并启动一个带有新值的定时器周期，必须先禁用并再次使能定时器。

2）当前定时器值寄存器(PIT_CVALn)

D31～D0 为当前计时器的值（TVL）。如果定时器是使能的，TVL 表示当前定时器的值。注意：如果计时器是禁用的，不要使用 TVL 字段，因为它的值是不可靠的。定时器使用减法计数器。如果 MCR[FRZ]位置 1，在调试模式下定时器值被冻结。

7.4.2　PIT 构件设计及测试实例

在本程序中，将 MCU 的串口与 PC 相连，PIT 每次中断进行一次计时，并通过串口将计时信息发送给 PC。PIT 模块具有初始化、使能 PIT 通道、禁止 PIT 通道及 PIT 中断处理函数。按照构件的思想，可将它们封装成独立的功能函数。初始化完成对 PIT 模块的工作属性的设定。对 PIT 进行编程，实际上已经涉及对硬件底层寄存器的直接操作，因此可以将初始化、启动和关闭等基本操作所对应的功能函数共同定义在命名为 pit.c 的文件中，并按照相对严格的构件设计原则对其进行封装，同时配以命名为 pit.h 的头文件，用来定义模块的基本信息和对外接口。将中断函数定义在名为 isr.c 的文件中。

下面通过封装 PIT 基本功能函数来进一步深刻理解 PIT 构件化编程思想。

1．PIT 构件头文件（pit.h）

```
//=============================================================================
//文件名称：pit.h
//功能概要：PIT 底层驱动程序头文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-5-8  V2.0
//=============================================================================

#ifndef PIT_H
#define PIT_H
//1 头文件
#include "common.h"
#define PIT_CH0_IRQ_NO 22
#define PIT_CH1_IRQ_NO 23
//2 宏定义
#define PIT_CH0 0
#define PIT_CH1 1
//=============================================================================
//函数名称:pit_init
```

```
//函数返回:无
//参数说明:pitno:表示 PIT 通道号，0,1
//        int_ms:表示中断的毫秒，可以从 1 到 2000000
//功能概要:设置相关寄存器的值
//=============================================================================
void pit_init(uint_8 pitno,uint_32 int_ms);
//=============================================================================
//函数名称:enable_pit_int
//函数返回:无
//参数说明:pitno:表示 PIT 通道号，0,1
//功能概要:开 PIT 中断
//=============================================================================
void enable_pit_int(uint_8 pitno);
//=============================================================================
//函数名称:disable_pit_int
//函数返回:无
//参数说明:pitno:表示 PIT 通道号，0,1
//功能概要:关闭 PIT 中断
//=============================================================================
void disable_pit_int(uint_8 pitno);

#endif
```

2．PIT 构件源文件（pit.c）

```
//=============================================================================
//文件名称: pit.c
//功能概要: pit 底层驱动程序文件
//版权所有: 苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新: 2015-5-8  V2.0
//=============================================================================
#include "pit.h"
//=============================================================================
//函数名称:pit_init
//函数返回:无
//参数说明:pitno:表示 PIT 通道号，0,1
//        int_ms:表示中断的毫秒，可以从 1 到 2000000
//功能概要:设置相关寄存器的值
//=============================================================================
void pit_init(uint_8 pitno,uint_32 int_ms)
{
    uint_32 mod_value;
    SIM_SCGC |= SIM_SCGC_PIT_MASK;              //使能 PIT 时钟门
    PIT_MCR &= ～PIT_MCR_MDIS_MASK;             //使能 PIT 模块

    disable_pit_int(pitno);

    //配置 PIT 模块
```

```
    mod_value = int_ms * BUS_CLK_KHZ;
    PIT_LDVAL(pitno) = mod_value;
    PIT_TCTRL(pitno) |= (PIT_TCTRL_TIE_MASK);  //开 PIT 模块中断
    PIT_TCTRL(pitno) |= (PIT_TCTRL_TEN_MASK);  //使能 PIT 模块运行

    enable_pit_int(pitno);
}

//=========================================================================
//函数名称:enable_pit_int
//函数返回:无
//参数说明:pitno:表示 PIT 通道号，0,1
//功能概要:开 PIT 中断
//=========================================================================
void enable_pit_int(uint_8 pitno)
{
    NVIC_EnableIRQ(PIT_CH0_IRQ_NO + pitno);    //开接收引脚的 IRQ 中断
}

//=========================================================================
//函数名称:disable_pit_int
//函数返回:无
//参数说明:pitno:表示 PIT 通道号，0,1
//功能概要:关闭 PIT 中断
//=========================================================================
void disable_pit_int(uint_8 pitno)
{
    NVIC_DisableIRQ(PIT_CH0_IRQ_NO + pitno);   //关接收引脚的 IRQ 中断
}
```

3．PIT 测试工程

测试工程功能概述如下。

（1）串口通信格式：波特率 9600，1 位停止位，无校验。

（2）PIT 使用总线时钟 24 MHz。

（3）上电或按复位按钮时，调试串口输出“Hello PIT!”。

（4）PIT 每 1 s 中断一次，在中断里 LIGHT_0～LIGHT_3 小灯状态改变（LIGHT_0～LIGHT_3 初始化时 2 亮 2 暗），同时调试串口输出“pit_ch0 isr”。

（5）PC 向 MCU 发送数据时，MCU 进入串口接收中断，将接收的一个字节直接回发。三个串口具有相同功能。

PIT 构件的测试工程位于网上光盘中的“..\program\CH07-KEA128-PIT”文件夹。

第 8 章　Flex 定时器 FTM

本章导读：本章阐述 KEA128 的 FTM 模块的构件化编程，包括基本定时功能、脉宽调制器（PWM）、输出比较功能和输入捕捉功能。主要内容有：8.1 节在简要阐述 FTM 基本知识的基础上，给出 KEA128 芯片 FTM 的技术要点及寄存器总览；8.2 节给出 FTM 的编程结构和基本定时测试实例；8.3 节给出 FTM 的 PWM 测试实例；8.4 节给出 FTM 的输出比较测试实例；8.5 节给出了 FTM 的输入捕捉测试实例。期望通过这一章的学习，读者能掌握 FTM 的原理。

本章参考资料：本章参考自《KEA128 参考手册》的第 26 章（FTM）。

8.1　FTM 基本知识

8.1.1　FTM 概述

Flex 定时器（FlexTimer，FTM）是飞思卡尔在早期 HCS08 定时器技术基础上，扩展功能来满足电机控制、智能照明和电源转换等技术需要而形成的一个具有包含基本定时、PWM、输出比较及输入捕捉功能的综合定时器[①]。

所有与 TPM 模块相同特性的寄存器完全兼容，Flex 定时器在相同的内核平台下的代码不需要任何改变就可以实现相同的功能。

与 TPM 相比增加了如下特性：有符号的递增计数器、硬件死区时间插入、错误控制输入、增强的触发功能、初始化和极性控制功能。专用的寄存器集增加了发动机控制和电源转换控制等，默认情况下为关闭这些新特性。通常由一组寄存器控制每一个特性。

Flex 定时器输入触发可以来自于比较器、ADC 或者其他子模块，具有自动初始化定时器功能。多个 Flex 定时器的初始化、输入时钟、初始和最终计数值都相同，并且计数器递增步调一致，这样它们实现同步可以提供一个更大的定时器。

FTM 模块结构可参考 KEA128 芯片手册第 371 页。

8.1.2　FTM 技术要点

本小节具体阐述 FTM 各个模块的技术要点，由于版面有限，这里只列举了常用的几个模块特性。

1．计数时钟源与分频

FTM 时钟源是可选的，时钟源可以是系统时钟、固定频率时钟或外部时钟；固定频率时

① 有些文献解释为：柔性定时器、弹性定时器或灵活定时器，本文直接使用 Flex 一词。

钟是一个附加的时钟输入，允许选择一个片上时钟源，而不是系统时钟；选择外部时钟源，FTM 时钟与输入引脚相连，因此需要同步 FTM 计数器和外部时钟源。

FTM 状态和控制寄存器（FTMx_SC）的 CLKS[1:0]位用于选择 FTM 计数器的时钟源。CLKS[1:0]=00 表示没有选择任何时钟，相当于关闭 FTM 计数器；CLKS[1:0]=01 表示选择系统时钟源；CLKS[1:0]=10，固定频率时钟；CLKS[1:0]=11，外部时钟。MCU 复位后，CLKS[1:0]=00，关闭 FTM 计数器。选择的时钟源的分频因子由 FTM 状态和控制寄存器（FTMx_SC)的 PS[2:0]位决定，PS[2:0]=000 表示 1 分频，PS[2:0]=001 表示 2 分频，PS[2:0]=010 表示 4 分频，PS[2:0]=011 表示 8 分频，PS[2:0]=100 表示 16 分频，PS[2:0]=101 表示 32 分频。

2．计数器

FTM 有一个 16 位计数器，供通道使用，计数器的操作主要有向上计数与向上向下计数。

增 1 计数：当 FTM 状态和控制寄存器（FTMx_SC）的中央对齐 PWM 选择位 CPWMS=0 时向上计数，CNTIN 定义了计数起始值，MOD 定义了计数的最终值。FTM 计数器先加载 CNTIN 值，然后开始计数，直到达到 MOD 值，然后计数器重新加载 CNTIN 值，重复以上步骤。如果 CNTIN=0x0000，FTM 计数和 TPM 向上计数一样，即向上无符号计数，如果 CNTIN[15]=1，FTM 是向上有符号计数，如果 CNTIN（15）=0 且 CNTIN=0x0000，FTM 也是向上无符号计数。

增/减 1 计数：当 CPWMS=1 时，向上向下计数。CNTIN 定义了计数起始值，MOD 定义了计数的最终值。同向上计数一样，FTM 计数器先加载 CNTIN 值，然后开始增 1 计数，直到达到 MOD 值，然后计数器执行减 1 操作，直到它回到 CNTIN 值，上下计数器重新开始。

3．通道功能

每个通道可以配置为输入捕捉、输出比较或边沿对齐 PWM 模式。处于输入捕捉模式时，捕捉可以发生在上升沿、下降沿或是沿跳变；几个通道可选择输入过滤。当处于输出比较模式时，输出信号可以被置 1、清 0 或在匹配时触发。所有通道可以配置为中央对齐 PWM 模式。每一对通道可以联合起来产生一个 PWM 信号，独立控制 PWM 信号的两种边沿。

FTM 模块除了定时器功能外，具有三个常用的模块：脉宽调制器（PWM）、输出比较和输入捕捉。这三个模块将分别在 8.3 节、8.4 节和 8.5 节进行说明。

8.1.3 FTM 寄存器总览

表 8-1 给出了 FTM 模块所有寄存器的基本介绍。

表 8-1 FTM 寄存器介绍

寄存器名称	功 能 描 述
状态和控制寄存器（FTMx_SC）	每个 FTM 模块中包含一个状态和控制寄存器，该寄存器里面包含计数器溢出标志，溢出中断允许设置，计数模式设置，时钟源选择和分频设置
计数器寄存器（FTMx_CNT）	该寄存器包含 FTM 计数器的值。复位时该寄存器清 0，向该寄存器写入任何值将会使该寄存器回到初始设定值。CNTIN 中保存的是初始设定值
模数寄存器（FTMx_MOD）	该寄存器保存 FTM 计数器的模数值，即计数器计数终止值，当计数器到达对应的模数值时，TOF 将在下一个时钟到来时置 1。此时计数器的值取决于选择的计数器计数方案，默认回到初始值

续表

寄存器名称	功 能 描 述
通道 n 状态和控制寄存器（FTMx_CnSC）	这是一个非常重要的寄存器，每个通道都有一个 CnSC 寄存器。该寄存器中包含通道中断标志位、中断使能控制位、通道设置及引脚功能设置
通道 n 值寄存器（FTMx_CnV）	每个通道都有一个 CnV 寄存器，在输入捕捉模式下，当捕捉到设置的边沿时，此时 FTM 计数器的值自动保存到 CnV 寄存器中，该值可用于反映捕捉事件发生的时刻。在输出模式下 CnV 寄存器保存输出匹配值，该值用于和 FTM 计数器的值进行比较，当相等时，则比较成功
计数器初始值寄存器（FTMx_CNTIN）	CNTIN 寄存器包括 FTM 计数器的初始值
特性模式选择寄存器（FTMx_MODE）	该寄存器主要设置错误中断、错误控制、捕捉测试模式、PWM 同步、写保护、通道输出初始化、FTM 增强特性使能，这些控制和所有通道都有关
输出掩码寄存器（FTMx_OUTMASK）	该寄存器为每个 FTM 通道提供了掩码。通道的掩码用于决定当匹配发生时，输出是否响应。这个特性可用于 BLDC 控制，PWM 信号在特定的时间提供一个电动机以实现电子整流
捕捉和比较状态寄存器（FTMx_STATUS）	该寄存器中包含了每个通道的 FTMx_CSC 寄存器中的 CHnF 位的拷贝以方便编程。这样一次就可以读出一个 FTM 模块的所有通道的标志位，读取后，写 0 清除
链接通道功能寄存器（FTMx_COMBINE）	该寄存器包含：错误控制、同步、死区插入、双边沿捕捉模式、补偿、双通道联合等功能。该寄存器在使用双通道联合功能时，主要是非对称 PWM 输出、双通道互补输出、双边沿捕捉等功能时需要合理设置
同步寄存器（FTMx_SYNC）	该寄存器用于设置 PWM 同步。一个同步事件能够执行 MOD、CV 和 OUTMASK 寄存器的同步，即使用缓冲器中的值更新这几个寄存器，这时 FTM 计数器也可以重新初始化。当 FTMEN=1 时，该寄存器必须合理设置
死区时间插入控制寄存（FTMx_DEADTIME）	这个寄存器选择死区分频器分频系数和死区时间值，所有的 FTM 通道使用该寄存器设置的时钟预分频值及死区时间用于死区插入
FTM 外部触发器寄存器（FTMx_EXTTRIG）	该寄存器表示：产生了通道触发；在 FTM 计数器等于其初始值时使能产生触发；并且选择使用哪些通道产生触发。在一个 PWM 周期，可以选择几个通道来产生多触发器。不使用通道 6 与通道 7 产生通道触发器
通道极性寄存器（FTMx_POL）	该寄存器定义了 FTM 通道的输出极性
错误模式状态(FTMx_FMS)	该寄存器包括错误检测标志位、写保护使能位和使能错误输入的逻辑或
输入捕捉过滤控制寄存器（FTMx_FILTER）	该寄存器选择输入通道的过滤值。通道 4，5，6，7 没有输入过滤
错误控制寄存器（FTMx_FLTCTRL）	该寄存器选择错误输入的过滤器的值，使能错误输入与错误输入过滤器
配置寄存器（FTMx_CONF）	寄存器选择时间号，在 TOF 置位前 FTM 计数器溢出，FTM 工作在 DEBUG 模式，使用一个外部的通用时间参考与使能全局时间基准信号
FTM 错误输入极性寄存器（FTMx_FLTPOL）	该寄存器定义错误输入极性
同步配置寄存器（FTMx_SYNCONF）	当检测到硬件触发器 j（j=0，1，2）时，FTM 清 TRIGj 位，同步配置寄存器选择 PWM 同步配置、SWOCTRL、INVCTRL 和 CNTIN 寄存器同步
FTM 反转控制寄存器（FTMx_INVCTRL）	该寄存器控制通道 n 输出何时变成通道 n+1 输出，通道 n+1 输出何时变成通道 n 输出。每一个 INVmEN 位为相应通道对 m 使能反转操作。该寄存器有一个写缓冲区，通过 INVCTRL 寄存器同步更新 INVmEN 位
FTM 软件输出控制寄存器（FTMx_SWOCTRL）	该寄存器使能通道 n 输出的软件控制并且定义强制通道 n 输出的值：ChnOC 位通过软件使能对应通道 n 输出的控制；CHnOCV 选择强制通道 n 输出的值
FTM PWM 加载寄存器（FTMx_PWMLOAD）	当 FTM 计数器把 MOD 寄存器值作为它下一个值或当一个通道(j)匹配发生时，使能 MOD、CNTIN、C(n)V 和 C(n+1)V 寄存器更新为写缓冲的值。当 FTM 计数器=C(j)V 时，通道(j)发生匹配



8.2 FTM 基本定时的编程结构与测试实例

8.2.1 FTM 基本定时的编程结构

以下寄存器的用法在 KEA128 的芯片手册上有详细的说明，下面按 FTM 初始化顺序阐述基本中断编程需要使用的寄存器。下面所列寄存器名中的 *x* 表示 FTM 模块编号，取 0～2；*n* 代表通道号，取值 0～5。

1．寄存器地址分析

KEA128 芯片有 3 个 FTM 模块。每个模块有其对应的寄存器。以下地址分析均为十六进制，为书写简化起见，在不至引起歧义的情况下，略去十六进制的前缀“0x”不写。

FTM 模块寄存器的地址= 4003_8000+*x*×1000+offset，其中，*x*=0～2，offset 各个寄存器的地址偏移。

2．状态和控制寄存器（FTMx_SC）

FTM 状态和控制寄存器，包括溢出状态标志位、用来配置中断的使能位、FTM 配置位、时钟源和预分频因子的控制位。这些控制与 FTM 模块的所有通道相关，其中 8～31 位保留。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	TOF	TOIE	CPWMS	CLKS		PS		
复位	0							

D7—定时器溢出标志。当 FTM 计数器超出 MOD 寄存器中的值时被硬件置位。当 TOF 被置位时，读 SC 寄存器，然后将 TOF 位写 0 会清除 TOF。将 TOF 写 1 没有影响。如果在读和写操作之间发生另一个 FTM 溢出事件，写操作将没有效果。因此，TOF 仍然是置位的，指明产生溢出。在这种情况下，因为先前的 TOF 的清除序列而导致中断请求没有丢失。

D6—TOIE 为定时器溢出中断允许位（Timer Overflow Interrupt Enable Bit）。该位用来设置是否允许定时器溢出中断。TOIE=1，允许定时器溢出中断；TOIE=0，不允许定时器溢出中断。该位可读写，读取的数值就是该位的实际值。

D5—CPWMS 为中心对齐 PWM 选择位（Center-Aligned PWM Select Bit）。对于输入捕捉、输出比较和边沿对齐排列 PWM 功能，FTM 以增 1 计数模式运行。对于中心对齐 PWM 功能，设置 CPWMS 可将 FTM 重新配置为以增/减 1 计数模式运行。CPWMS=0，通过对每个通道的状态和控制寄存器的 MSnB:MSnA 进行选择，所有 FTM 通道运行为输入捕捉、输出比较或边沿对齐 PWM 模式；CPWMS=1，所有 FTM 通道以中心对齐 PWM 模式运行。

D4～D3—CLKS 为时钟源选择。从三个 FTM 计数器时钟源中选择一个，这个域是写保护的，当 ODE[WPDIS]=1 时，它才能被写。00 没有选择任何时钟（实际上是禁止 FTM 计数器）；01 选择系统时钟；10 选择固定频率时钟；11 选择外部时钟。

D2～D0—为 PS 预分频因子选择。为由 CLKS 选择的时钟源选择一个分频因子。这个预分频因子在新值更新到寄存器位后的下一个系统时钟周期开始起作用。这个域是写保护的，

当 MODE[WPDIS]=1 时，它只能被写。000 除以 1；001 除以 2；010 除以 4；011 除以 8；100 除以 16；101 除以 32；110 除以 64；111 除以 128。

3．通道 *n* 状态和控制寄存器（FTMx_CnSC）

CnSC 包含了通道中断状态标志控制位，用于配置中断使能、通道配置和引脚功能，其中 8～31 位预留。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	CHF	CHIE	MSB	MSA	ELSB	ELSA	预留	预留
复位	0							

D7—CHnF 为通道标志。当通道有事件触发时，此位被硬件置位。当 CHnF 被置位时，读 CSC 寄存器可以清 CHF。写 1 无效。如果在读写操作时发生另一个事件，写操作无效；因此通道事件发生时，CHF 不会变。CHF =0 无通道事件发生；CHF =1 有通道事件发生。

D6—CHnIE 为通道中断允许位（Channel n Interrupt Enable Bit）。该位用来设置是否允许发生输入捕捉中断。该位置 1，在发生输入捕捉时会产生中断，在中断服务程序中可以进行相关的处理，通常这个中断服务程序中应包含清除通道标志位的指令。CHnIE=0，允许通道 *n* 中断；CHnIE=1，禁止通道 *n* 中断。

D5～D4—MSnB～MSnA 为通道 *n* 模式选择位（Mode Select Bit）。每一个定时器都可以工作在输入捕捉，输出比较或 PWM 输出模式。这两位用来选择工作模式。模式选择位和下面的跳变沿/输出电平选择位共同决定通道的工作方式，见表 8-2。

表 8-2　模式、边沿和级别选择

DECAPEN	COMBINE	CPWMS	MSnB:MSnA	ELSnB:ELSnA	模　式	配　置
X	X	X	XX	00	引脚不用于 FTM，恢复通道引脚为通用 I/O 或其他外围设备的控制	
0	0	0	00	01	输入捕捉	
				10		下降沿捕捉
				11		沿跳变
			01	01	输出比较	匹配时输出取反
				10		匹配时输出清 0
				11		匹配时输出置 1
			1X	10	边沿对齐 PWM	正极性
				X1		负极性
		1	XX	10	中心对齐 PWM	正极性
				X1		负极性
	1	0	XX	10	混合 PWM	正极性
				X1		负极性
1	0	0	X0	见表 8-3	沿跳变捕捉	单次捕捉模式
			X1			连续捕捉模式

D3～D2—ELSnB～ELSnA 为跳变沿/输出电平选择位（Edge/Level Select Bit）。在输入捕捉时可以设定上升沿或下降沿捕捉、输出比较也可以设定输出高或低电平。这就需要设定跳变沿/输出电平选择位。每一位的设定和工作方式的关系，见表 8-3。

D1～D0—预留，只读为 0。

表 8-3 沿跳变捕捉模式-边沿极性选择

ELSnB	ELSnA	通 道 使 能	边 沿 检 测
0	0	禁止	无
0	1	使能	上升沿
1	0	使能	下降沿
1	1	使能	上升沿和下降沿

4．计数寄存器（FTMx_CNT）

FTMx_CNT 是一个 32 位寄存器，分为高 16 位、低 16 位，其中高 16 位预留。该寄存器的作用是：当定时器的状态和控制寄存器的 TSTOP 位=0 时，即允许计数，每一计数周期，其值自动加 1，当它达到设定值(在 16 位预置寄存器中)时，TOF=1，同时计数寄存器自动清 0。复位时，计数寄存器的初值为 0。

5．通道 *n* 数值寄存器（FTMx_CnV）

由于通道数值寄存器是一个 32 位的寄存器，分为高字节和低字节，在写入时，通常会分两次写入。这时需要注意，在写入通道数值寄存器的高位字节后，为了防止此时的计数恰好等于计数寄存器的值，而产生不希望的输出比较动作，输出比较模块此时是被禁止的。只有在继续写入通道数值寄存器的低位字节后，输出比较模块才开始和计数寄存器进行比较。所以，输出比较功能的开始，应该是对寄存器两个字节的写入结束后。通道数值寄存器和标志位在复位时被清零，在初始化输出比较功能时应该小心谨慎，一般采用以下的步骤。

（1）写入通道数值寄存器的高位字节，禁止输出比较功能。

（2）取状态寄存器，清除 CHnF 位。

（3）写入通道数值寄存器低位字节，使输出比较功能工作。

8.2.2 FTM 基本定时构件与测试实例

为了理解 FTM 的基本定时功能，此处给出了利用 FTM 定时器溢出中断完成基本定时的测试程序。通过 FTM 的基本定时功能，每 1 s 改变小灯的状态。通过本测试实例，将会深刻理解 FTM 的基本定时功能。

按照构件的思想，可将它们封装成独立的功能函数，初始化完成对 FTM 模块的工作属性的设定。对 FTM 进行编程，实际上已经涉及对硬件底层寄存器的直接操作，因此可以将初始化、启动和关闭等基本操作所对应的功能函数共同定义在命名为 ftm.c 的文件中，并按照相对严格的构件设计原则对其进行封装，同时配以命名为 ftm.h 的头文件，用来定义模块的基本信息和对外接口。将中断函数定义在名为 isr.c 的文件中。

1．FTM 基本定时驱动构件头文件（ftm.h）

```
//=============================================================================
//文件名称：ftm.h
//功能概要：FTM 基本定时驱动构件源文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
```

```
//版本更新：2015-05-11  V1.0
//芯片类型：KEA128
//============================================================================
#include "common.h"        //包含公共要素头文件

#ifndef FTM_H_
#define FTM_H_

#define FTM_0 0
#define FTM_1 1
#define FTM_2 2

#define FTM0_irq_no 17
#define FTM1_irq_no 18
#define FTM2_irq_no 19

//============================================================================
//函数名称：FTM_Init
//函数返回：无
//参数说明：ftmNo:模块号
//         period：中断周期
//功能概要：FTM 基本定时初始化
//============================================================================
void FTM_Init(uint_8 ftmNo,uint_32 period);

//============================================================================
//函数名称：enable_FTM_int
//函数返回：无
//参数说明：ftmNo:FTM 模块号
//功能概要：开 FTM 定时中断
//============================================================================
void enable_FTM_int( uint_8 ftmNo);

//============================================================================
//函数名称：disable_FTM_int
//函数返回：无
//参数说明：ftmNo:FTM 模块号
//功能概要：关 FTM 定时中断
//============================================================================
void disable_FTM_int(uint_8 ftmNo);

#endif /*FTM_H_*/
```

2．FTM 基本定时驱动构件源文件（ftm.c）

```
//============================================================================
//文件名称：ftm.c
//功能概要：FTM 基本定时驱动构件源文件
```

```
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-05-11  V1.0
//芯片类型：KEA128
//=============================================================================
#include "ftm.h"

//FTM0、FTM1、FTM2 地址映射
const FTM_MemMapPtr FTM_ARR[]={FTM0_BASE_PTR,FTM1_BASE_PTR,FTM2_BASE_PTR};

//----------------------以下为内部函数存放处-------------------------------
//=============================================================================
//函数名称：FTM_Init
//函数返回：无
//参数说明：ftmNo:模块号
//         period：中断周期
//功能概要：FTM 基本定时初始化
//=============================================================================
void FTM_Init(uint_8 ftmNo,uint_32 period)
{
    switch(ftmNo)
    {
    case FTM_0:
        //使能 FTM0 时钟
        SIM_SCGC |= SIM_SCGC_FTM0_MASK;
        //总线时钟频率 24 MHz，分频因子为 128，分频后位 187.5 kHZ
        FTM0_SC |= FTM_SC_PS(7);
        //使能中断
        FTM0_C0SC |= FTM_CnSC_CHIE_MASK;
        //设置模式
        FTM0_C0SC |= FTM_CnSC_MSA_MASK;
        //计数器清 0
        FTM0_CNT = 0;
        //设置周期
        FTM0_C0V =period;
        //FTM0 使用系统时钟
        FTM0_SC |= FTM_SC_CLKS(1);
        break;
    case FTM_1:
        //使能 FTM1 时钟
        SIM_SCGC |= SIM_SCGC_FTM1_MASK;
        //总线时钟频率 24 MHz，分频因子为 128，分频后位 187.5 kHZ
        FTM1_SC |= FTM_SC_PS(7);
        //使能中断
        FTM1_C0SC |= FTM_CnSC_CHIE_MASK;
        //设置模式
        FTM1_C0SC |= FTM_CnSC_MSA_MASK;
        //计数器清 0
```

```
        FTM1_CNT = 0;
        //设置周期
        FTM1_C0V =period;
        //FTM1 使用系统时钟
        FTM1_SC |= FTM_SC_CLKS(1);
        break;
    case FTM_2:
        //使能 FTM2 时钟
        SIM_SCGC |= SIM_SCGC_FTM2_MASK;
        //总线时钟频率 24 MHz，分频因子为 128，分频后位 187.5 kHz
        FTM2_SC |= FTM_SC_PS(7);
        //使能中断
        FTM2_C0SC |= FTM_CnSC_CHIE_MASK;
        //设置模式
        FTM2_C0SC |= FTM_CnSC_MSA_MASK;
        //计数器清 0
        FTM2_CNT = 0;
        //设置周期
        FTM2_C0V = period;
        //FTM2 使用系统时钟
        FTM2_SC |= FTM_SC_CLKS(1); //FTM2 使用系统时钟
    }

}

//=========================================================================
//函数名称：enable_FTM_int
//函数返回：无
//参数说明：ftmNo:FTM 模块号
//功能概要：开 FTM 定时中断
//=========================================================================
void enable_FTM_int( uint_8 ftmNo)
{
    //开 FTM 中断
    NVIC_EnableIRQ(FTM0_irq_no + ftmNo);
}

//=========================================================================
//函数名称：disable_FTM_int
//函数返回：无
//参数说明：ftmNo:FTM 模块号
//功能概要：关 FTM 定时中断
//=========================================================================
void disable_FTM_int(uint_8 ftmNo)
{
    //关 FTM 中断
    NVIC_DisableIRQ(FTM0_irq_no + ftmNo);
}
```

3．FTM 基本定时头文件（main.c）

```
//说明见工程文件夹下的 Doc 文件夹内 Readme.txt 文件
#define GLOBLE_VAR          //只需在 main.c 中定义一次，用来防止全局变量的重复定义

#include "includes.h"

int main(void)
 {
    //1. 声明主函数使用的变量
    //2. 关总中断
    DISABLE_INTERRUPTS;                         //关总中断
    //3. 初始化外设模块
    uart_init (UART_Debug,9600);                //初始化"调试串口"
    uart_init (UART_0,9600);
    uart_init (UART_1,9600);
    uart_init (UART_2,9600);
    light_init(LIGHT_0, LIGHT_OFF);             //初始化灯 0
    light_init(LIGHT_1, LIGHT_OFF);             //初始化灯 1
    light_init(LIGHT_2, LIGHT_OFF);             //初始化灯 2
    light_init(LIGHT_3, LIGHT_OFF);             //初始化灯 3
    FTM_Init(FTM_2,37500);  //FTM2 基本定时初始化，37500 对应定时中断周期 100 ms
    printf("Hello FTM_Timer!\r\n");             //串口发送初始化提示
    //4. main 中使用的局部变量初始化
    //5. 全局变量初始化
    FTMCounter=0;                               //FTM 中断次数计数器初始化
    //6. 使能模块中断
    uart_enable_re_int(UART_Debug);             //使能 UART_Debug 的接收中断
    uart_enable_re_int(UART_0);
    uart_enable_re_int(UART_1);
    uart_enable_re_int(UART_2);
    enable_FTM_int(FTM_2);                      //开启 FTM 中断
    //7. 开总中断
    ENABLE_INTERRUPTS;                          //开总中断

    //进入主循环
    //主循环开始============================================================
for (;;)
    {
        //运行指示灯闪烁-------------------------------------------------
        if(FTMCounter>=10)                  //计数 10 次中断，1 s 中改变四盏小灯的状态
        {
            FTMCounter=0;                   //FTM 中断计数器清 0
            light_change(LIGHT_0);          //灯 0（LIGHT_0）状态变化
            light_change(LIGHT_1);          //灯 1（LIGHT_1）状态变化
            light_change(LIGHT_2);          //灯 2（LIGHT_2）状态变化
            light_change(LIGHT_3);          //灯 3（LIGHT_3）状态变化
        }
```

```
        //以下加入用户程序-----------------------------------------------
    }  //主循环 end_for

    //主循环结束==================================================================
    return 0;
}
```

4．FTM 基本定时中断服务例程源文件（isr.c）

```
//FTM2 中断服务例程
void isr_ftm2()
{
    (void)FTM2_C0SC;                          //读 CnSC 寄存器
    FTM2_C0SC ^= FTM_CnSC_CHF_MASK;           //清 CHF 位
    FTM2_CNT=0;
    FTMCounter++;
}
```

5．FTM 测试工程

测试工程功能概述如下。

（1）串口通信格式：波特率 9600，1 位停止位，无校验。

（2）上电或按复位按钮时，调试串口输出“Hello FTM_Timer!”。

（3）主循环中，通过 FTM2 通道 0 计数，100 ms 产生一次中断，每 10 次中断改变 LIGHT_0～LIGHT_3 小灯状态（整体效果为四盏灯同时闪烁）。

（4）PC 向 MCU 发送数据时，MCU 进入串口接收中断，将接收的一个字节直接回发。三个串口具有相同功能。

FTM 构件的测试工程位于网上光盘中的“..\program\CH08-KEA128-FTM_Timer”文件夹。

8.3　FTM 模块的脉宽调制（PWM）功能

8.3.1　脉宽调试器 PWM 基本工作原理

PWM 产生一个在高电平和低电平之间重复交替的输出信号，这个信号称为 PWM 信号，也叫做脉宽调制波。通过指定所需的时钟周期和占空比来控制高电平和低电平的持续时间。通常定义占空比为信号处于高电平的时间（或时钟周期数）占整个信号周期的百分比，方波的占空比是 50%。脉冲宽度是指脉冲处于高电平的时间。图 8-1 给出了几个不同占空比的示意图。图中给出 PWM 信号的周期是 8 个时钟周期 $T_{PWM}=8T_{CLK}$，图 8-1(a)中，PWM 的高电平为 $2T_{CLK}$，所以占空比=2/8=25%，图 8-1(b)、图 8-1(c)可以类似计算。

PWM 的常见应用是给其他设备产生类似于时钟的信号。例如，PWM 可用来控制灯以一定频率闪烁。PWM 的另一个常见用途是控制输入到某个设备的平均电流或电压。例如，一个直流电机在输入电压时会转动，而转速与平均输入电压的大小成正比，假设每分

钟转速（rpm）为输入电压的 100 倍，如果转速要达到 125 rpm，则需要 1.25 V 的平均输入电压；如果转速要达到 250 rpm，则需要 2.50 V 的平均输入电压。在图 8-1 中，如果逻辑 1 是 5 V，逻辑 0 是 0 V，则图 8-1(a)的平均电压是 1.25 V，图 8-1(b)的平均电压是 2.5V，图 8-1(c)的平均电压是 3.75 V。可见，利用 PWM 可以设置适当的占空比来得到所需的平均电压，如果所设置的周期足够小，电机就可以平稳运转（即不会明显感觉到电机在加速或减速）。

PWM 的另一种应用是控制命令字编码。例如，通过发送不同宽度的脉冲，代表不同含义。假如用此来控制无线遥控车，宽度 1 ms 代表左转命令，4 ms 代表右转命令，8 ms 代表前进命令。接收端可以使用定时器来测量脉冲宽度，在脉冲开始时启动定时器，脉冲结束时停止定时器，由此来确定所经过的时间，从而判断收到的命令。

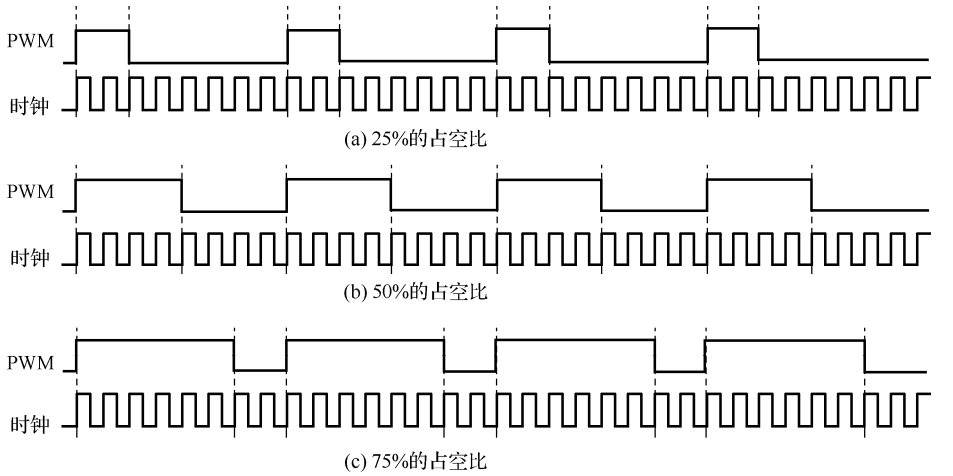


图 8-1 PWM 的占空比的计算方法

8.3.2 KEA128 的三种 PWM 模式

KEA128 的 PWM 有三种模式：边沿对齐 PWM、中心对齐 PWM 和通道合并模式。

1．边沿对齐 PWM（Edge-Aligned PWM）

所有 PWM 信号的边沿在周期的开始对齐，如图 8-2 所示。EPWM 周期由 MOD 寄存器和 CNTIN 寄存器的值决定（具体计算公式为 MOD – CNTIN + 0x0001），脉冲宽度（占空比）由 CnV 寄存器和 CNTIN 寄存器的值决定（具体计算格式为 CnV – CNTIN）。

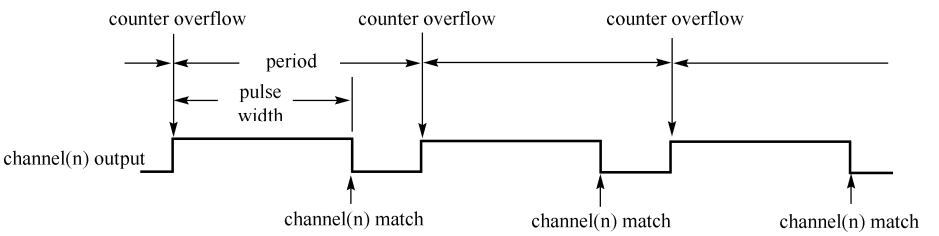


图 8-2 边沿对齐 PWM

2．中央对齐 PWM（Center-Aligned PWM）

如图 8-3 所示，CPWM 周期由 2×(MOD–CNTIN)决定，脉冲宽度（占空比）由 2×(CnV–CNTIN)决定，MOD 必须必须在 0x0001 到 0x7FFF 的范围内。

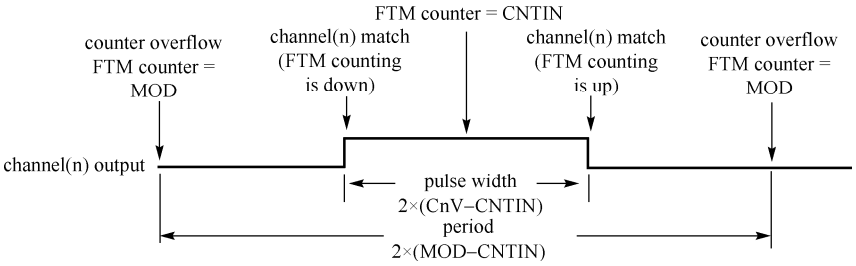


图 8-3　中央对齐 PWM

3．通道合并模式

通道 *n*（一个偶数通道）和通道 *n*+1（相邻的奇数通道）合并到通道 *n* 输出一路 PWM 信号，周期由（MOD–CNTIN+0x0001）决定，PWM 脉冲宽度（占空比）由（C(n+1)V–C(n)V）的绝对值决定。其中 C(*n*)V 为通道 *n* 的 CV 寄存器值。

8.3.3　FTM 引脚复用

80 引脚 KEA128 芯片共有三个 FTM 模块，分别是 FTM0、FTM1 和 FTM2，每个 FTM 模块的时钟源均使用总线时钟。FTM0 和 FTM1 各有两个通道，FTM2 有六个通道。通过系统集成模块（SIM）提供的引脚选择寄存器编程来设定引脚的 FTM 功能。根据附录 A（KEAZ128 引脚功能分配），可以配置为 FTM 的引脚见表 8-4。

表 8-4　KEA128 的 FTM 引脚

引　脚　号	引　脚　名	ALT2	ALT3	ALT4
1	PTD1	FTM2_CH3		
2	PTD0	FTM2_CH2		
6	PTE7			FTM1_CH1
7	PTH2			FTM1_CH0
18	PTH1	FTM2_CH1		
19	PTH0	FTM2_CH0		
22	PTB5	FTM2_CH5		
23	PTB4	FTM2_CH4	PWT_IN1	ADC0_SE4
24	PTC3	FTM2_CH3		
25	PTC2	FTM2_CH2		
31	PTC1	FTM2_CH1		
32	PTC0	FTM2_CH1		
39	PTB3		FTM0_CH1	
40	PTB2		FTM0_CH1	
50	PTG7	FTM2_CH5		
51	PTG6	FTM2_CH4		

续表

引 脚 号	引 脚 名	ALT2	ALT3	ALT4
52	PTG5	FTM2_CH3		
53	PTG4	FTM2_CH2		
54	PTF1	FTM2_CH1		
55	PTF0	FTM2_CH0		
61	PTA1	FTM0_CH1		
62	PTA0	FTM0_CH0		

8.3.4 PWM 构件与测试实例

本节给出了 PWM 的一个测试实例，将 PTC0 配置成 FTM2 通道 0，通过示波器可以看到该引脚输出的 PWM 波形。另外，PTC0 与小灯相连，主函数中通过不断地改变占空比，控制该小灯的亮度逐渐变化。在测试实例中，选择 24 MHz 的总线时钟，并进行 16 分频，分频后得到的时钟频率为 1.5 MHz。PWM 初始化函数中传入的参数 f_{req} 可以改变 PWM 的周期，duty 可改变 PWM 的占空比，且占空比的精度可达 $f_{req}/1500$。

按照构件的思想，可将它们封装成独立的功能函数，初始化完成对 PWM 模块的工作属性的设定。对 PWM 进行编程，实际上已经涉及对硬件底层寄存器的直接操作，因此可以将初始化、启动和关闭等基本操作所对应的功能函数共同定义在命名为 pwm.c 的文件中，并按照相对严格的构件设计原则对其进行封装，同时配以命名为 pwm.h 的头文件，用来定义模块的基本信息和对外接口。

1．PWM 驱动构件头文件（pwm.h）

```
//=============================================================================
//文件名称：pwm.h
//功能概要：PWM 底层驱动构件源文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-05-11  V1.0
//芯片类型：KEA128
//=============================================================================
#include "common.h"                 //包含公共要素头文件

#ifndef FTM_H_
#define FTM_H_

//FTM 模块号
#define FTM_0 0
#define FTM_1 1
#define FTM_2 2

//通道号，FTM0 和 FTM1：通道 0～通道 1;FTM2：通道 0～通道 5
#define CHANNEL0 0
#define CHANNEL1 1
#define CHANNEL2 2
```

```
#define CHANNEL3 3
#define CHANNEL4 4
#define CHANNEL5 5

//------------------------FTM0 通道和引脚选择-------------------------------
//注：通过展开以下宏定义并修改宏定义值可选择通道号和引脚

//FTM0 通道 0 引脚组:1=PTA0 脚，2=PTB2 脚
//#define FTM0_CHANNEL0_GROUP 1    //SD-FSL-KL25-EVB 板上使用 PTA14～15 脚

//FTM0 通道 1 引脚组:1=PTA1 脚，2=PTB3 脚
//#define FTM0_CHANNEL1_GROUP 1
//--------------------------------------------------------------------------


//-------------------------FTM1 通道和引脚选择-----------------------------
//注：通过展开以下宏定义并修改宏定义值可选择通道号和引脚

//FTM1 通道 0 引脚组:1=PTC4 脚,2=PTH2 脚
//#define FTM1_CHANNEL0_GROUP 1

//FTM1 通道 1 引脚组:1=PTC5 脚,2=PTE7 脚
//#define FTM1_CHANNEL1_GROUP 1
//--------------------------------------------------------------------------


//-------------------------FTM2 通道和引脚选择-----------------------------
//注：通过展开以下宏定义并修改宏定义值可选择通道号和引脚

//FTM2 通道 0 引脚组: 1=PTC0 脚, 2=PTH0 脚, 3=PTF0 脚
#define FTM2_CHANNEL0_GROUP 1          //KEA128 板上使用 PTC0 脚

//FTM2 通道 1 引脚组: 1=PTC1 脚, 2=PTH1 脚, 3=PTF1 脚
//#define FTM2_CHANNEL1_GROUP 1

//FTM2 通道 2 引脚组: 1=PTC2 脚, 2=PTD0 脚, 3=PTG4 脚
//#define FTM2_CHANNEL2_GROUP 1

//FTM2 通道 3 引脚组: 1=PTC3 脚, 2=PTD1 脚, 3=PTG5 脚
//#define FTM2_CHANNEL3_GROUP 1

//FTM2 通道 4 引脚组: 1=PTB4 脚, 2=PTG6 脚
//#define FTM2_CHANNEL4_GROUP 1

//FTM2 通道 5 引脚组: 1=PTB5 脚, 2=PTG7 脚
//#define FTM2_CHANNEL5_GROUP  1       //KEA128 板上使用 PTB5 脚
//--------------------------------------------------------------------------
```

```
//=============================================================================
//函数名称：FTM_PWM_init
//函数返回：无
//参数说明：ftmNo:FTM 模块号，FTM_0、FTM_1、FTM_2
//         channel:选择通道号，FTM0 和 FTM1：通道 0、通道 1
//                              FTM2:通道 0、1、2、3、4、5
//         freq：设置 PWM 的频率，单位为 KHz。取值从 0.012 到 1500
//         duty：占空比，精度取决于设置的 freq 值，精度为 freq/1500。
//功能概要：初始化 PWM 模块
//=============================================================================
void FTM_PWM_init(uint_8 ftmNo,uint_8 channel, float freq, float duty);

//=============================================================================
//函数名称：FTM_PWM_set_duty
//函数返回：无
//参数说明：ftmNo:FTM 模块号，FTM_0、FTM_1、FTM_2
//         channel:选择通道号，FTM0 和 FTM1：通道 0、通道 1
//                              FTM2:通道 0、1、2、3、4、5
//         duty：占空比，精度取决于设置的 mod 值，精度为 1/mod。
//功能概要：初始化 PWM 模块
//=============================================================================
void FTM_PWM_set_duty(uint_8 ftmNo, uint_8 channel, float duty);

#endif
```

2．PWM 驱动构件源程序文件（pwm.c）

```
//=============================================================================
//文件名称：pwm.c
//功能概要：PWM 底层驱动构件源文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-05-11  V1.0
//芯片类型：KEA128
//=============================================================================
#include "pwm.h"

//FTM0、FTM1、FTM2 地址映射
const FTM_MemMapPtr FTM_ARR[]={FTM0_BASE_PTR,FTM1_BASE_PTR,FTM2_BASE_PTR};

static uint_16 MOD_VALUE;

//=============================================================================
//函数名称：FTM_PWM_init
//函数返回：无
//参数说明：ftmNo:FTM 模块号,FTM_0、FTM_1、FTM_2
//         channel:选择通道号,FTM0 和 FTM1：通道 0、通道 1
//                              FTM2:通道 0、1、2、3、4、5
```

```
//          freq：设置 PWM 的频率，单位为 kHz，取值为 0.012 到 1500
//          duty：占空比，精度取决于设置的 freq 值，精度为 freq/1500。
//功能概要：初始化 PWM 模块
//=========================================================================
void FTM_PWM_init(uint_8 ftmNo,uint_8 channel, float freq, float duty)
{
    uint_16 duty_channel;
    uint_16 mod;

    //防止模块号越界
    if(ftmNo>2) ftmNo=2;

    //计算 mod 值，周期（ms）= MOD / 1500
    if(freq <= 0.012)
        mod = 0xFFFF;
    else if(freq>1500)
        mod=0;
    else
        mod = 1500 / freq;

    //防止周期越界
    if(duty > 100)
        duty = 100;
    if(duty < 0)
        duty = 0;

    switch(ftmNo)
    {
        case FTM_0:
            #if(FTM0_CHANNEL0_GROUP==1)
                //复用引脚 PTA0 为 FTM0 通道 0 功能
                SIM_PINSEL0 &= ~SIM_PINSEL_FTM0PS0_MASK;
            #endif

            #if(FTM0_CHANNEL0_GROUP==2)
                //复用引脚 PTB2 为 FTM0 通道 0 功能
                SIM_PINSEL0 |= SIM_PINSEL_FTM0PS0_MASK;
            #endif

            #if(FTM0_CHANNEL1_GROUP==1)
                //复用引脚 PTA1 为 FTM0 通道 1 功能
                SIM_PINSEL0 &= ~SIM_PINSEL_FTM0PS1_MASK;
            #endif

            #if(FTM0_CHANNEL1_GROUP==2)
                //复用引脚 PTB3 为 FTM0 通道 1 功能
                SIM_PINSEL0 |= SIM_PINSEL_FTM0PS1_MASK;
```

```
        #endif

        //使能 FTM0 的总线时钟
        SIM_SCGC |= SIM_SCGC_FTM0_MASK;
        break;

    case FTM_1:
        #if(FTM1_CHANNEL0_GROUP==1)
            //复用引脚 PTC4 为 FTM1 通道 0 功能
            SIM_PINSEL0 &= ~SIM_PINSEL_FTM1PS0_MASK;
        #endif

        #if(FTM1_CHANNEL0_GROUP==2)
            //复用引脚 PTH2 为 FTM1 通道 0 功能
            SIM_PINSEL0 |= SIM_PINSEL_FTM1PS0_MASK;
        #endif

        #if(FTM1_CHANNEL1_GROUP==1)
            //复用引脚 PTC5 为 FTM1 通道 1 功能
            SIM_PINSEL0 &= ~SIM_PINSEL_FTM1PS1_MASK;
        #endif

        #if(FTM1_CHANNEL1_GROUP==2)
            //复用引脚 PTE7 为 FTM1 通道 1 功能
            SIM_PINSEL0 |= SIM_PINSEL_FTM1PS1_MASK;
        #endif

        //使能 FTM1 的总线时钟
        SIM_SCGC |= SIM_SCGC_FTM1_MASK;
        break;

    case FTM_2:
        #if(FTM2_CHANNEL0_GROUP==1)
            //复用引脚 PTC0 为 FTM2 通道 0 功能
            SIM_PINSEL1 &= ~SIM_PINSEL1_FTM2PS0_MASK;
        #endif

        #if(FTM2_CHANNEL0_GROUP==2)
            //复用引脚 PTH0 为 FTM2 通道 0 功能
            SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS0(1);
        #endif

        #if(FTM2_CHANNEL0_GROUP==3)
            //复用引脚 PTF0 为 FTM2 通道 0 功能
            SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS0(2);
        #endif
```

```
#if(FTM2_CHANNEL1_GROUP==1)
    //复用引脚 PTC1 为 FTM2 通道 1 功能
    SIM_PINSEL1 &= ～SIM_PINSEL1_FTM2PS1_MASK;
#endif

#if(FTM2_CHANNEL1_GROUP==2)
    //复用引脚 PTH1 为 FTM2 通道 1 功能
    SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS1(1);
    #endif

 #if(FTM2_CHANNEL1_GROUP==3)
     //复用引脚 PTF1 为 FTM2 通道 1 功能
     SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS1(2);
  #endif

#if(FTM2_CHANNEL2_GROUP==1)
    //复用引脚 PTC2 为 FTM2 通道 2 功能
    SIM_PINSEL1 &= ～SIM_PINSEL1_FTM2PS2_MASK;
#endif

#if(FTM2_CHANNEL2_GROUP==2)
    //复用引脚 PTD0 为 FTM2 通道 2 功能
    SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS2(1);
 #endif

 #if(FTM2_CHANNEL2_GROUP==3)
     //复用引脚 PTG4 为 FTM2 通道 2 功能
     SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS2(2);
  #endif

#if(FTM2_CHANNEL3_GROUP==1)
    //复用引脚 PTC3 为 FTM2 通道 3 功能
    SIM_PINSEL1 &= ～SIM_PINSEL1_FTM2PS3_MASK;
#endif

#if(FTM2_CHANNEL3_GROUP==2)
    //复用引脚 PTD1 为 FTM2 通道 3 功能
    SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS3(1);
#endif

#if(FTM2_CHANNEL3_GROUP==3)
   //复用引脚 PTG5 为 FTM2 通道 3 功能
   SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS3(2);
#endif

#if(FTM2_CHANNEL4_GROUP==1)
    //复用引脚 PTB4 为 FTM2 通道 4 功能
```

```
            SIM_PINSEL1 &= ~SIM_PINSEL1_FTM2PS4_MASK;
        #endif

        #if(FTM2_CHANNEL4_GROUP==2)
            //复用引脚 PTG6 为 FTM2 通道 4 功能
            SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS4_MASK;
        #endif

        #if(FTM2_CHANNEL5_GROUP==1)
            //复用引脚 PTB5 为 FTM2 通道 5 功能
            SIM_PINSEL1 &= ~SIM_PINSEL1_FTM2PS5_MASK;
        #endif

        #if(FTM2_CHANNEL5_GROUP==2)
            //复用引脚 PTG7 为 FTM2 通道 5 功能
            SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS5_MASK;
        #endif

        //使能 FTM2 的总线时钟
        SIM_SCGC |= SIM_SCGC_FTM2_MASK;
        break;
    }

    //写保护禁止
    FTM_ARR[ftmNo]->MODE |= FTM_MODE_WPDIS_MASK;
    //总线时钟频率 24 MHz,分频因子 16,分频后为 1.5 MHz
    FTM_ARR[ftmNo]->SC |= FTM_SC_PS(4);
    //左对齐，先高电平后低电平
    FTM_CnSC_REG(FTM_ARR[ftmNo], channel) |= FTM_CnSC_ELSB_MASK;
    //边沿对齐
    FTM_CnSC_REG(FTM_ARR[ftmNo], channel) &= ~FTM_CnSC_ELSA_MASK;
    FTM_CnSC_REG(FTM_ARR[ftmNo], channel) |= FTM_CnSC_MSB_MASK;
    //计数器清零
    FTM_ARR[ftmNo]->CNT=0;
    //中断禁止
    FTM_ARR[ftmNo]->SC &=  ~FTM_SC_TOIE_MASK;
    //FTM2 使能
    FTM_ARR[ftmNo]->MODE &= ~FTM_MODE_FTMEN_MASK;
    //启用需要的输出通道，屏蔽其他通道
    FTM_ARR[ftmNo]->OUTMASK = (0xFE << channel);
    //禁止双边沿捕捉模式
    FTM_ARR[ftmNo]->COMBINE = 0;
    //FTM2 计数初始值为 0
    FTM_ARR[ftmNo]->CNTIN = 0;
    //周期=(MOD-CNTIN+1)*时钟周期=2.5ms
    FTM_ARR[ftmNo]->MOD = mod;
    MOD_VALUE = mod;
```

```
    //计算通道寄存器的值，设置占空比
    duty_channel = (uint_16)((float)mod * duty / 100);
    //设置通道值寄存器
    FTM_CnV_REG(FTM_ARR[ftmNo], channel) = duty_channel;
    //使用总线时钟
    FTM_ARR[ftmNo]->SC |= FTM_SC_CLKS(1);
}

//=========================================================================
//函数名称：FTM_PWM_set_duty
//函数返回：无
//参数说明：ftmNo:FTM 模块号,FTM_0、FTM_1、FTM_2
//         channel:选择通道号,FTM0 和 FTM1：通道 0、通道 1
//                             FTM2:通道 0、1、2、3、4、5
//         duty：占空比,精度取决于设置的 mod 值，精度为 1/mod
//功能概要：初始化 PWM 模块
//=========================================================================
void FTM_PWM_set_duty(uint_8 ftmNo, uint_8 channel, float duty)
{
    uint_16 duty_channel;

    if(duty > 100)
        duty = 100;
    if(duty < 0)
        duty = 0;
    //写保护禁止
    //FTM_ARR[ftmNo]->MODE |= FTM_MODE_WPDIS_MASK;
    //计算通道寄存器的值，设置占空比
    duty_channel = (uint_16)((float)MOD_VALUE * duty / 100);
    //设置通道值寄存器
    FTM_CnV_REG(FTM_ARR[ftmNo], channel) = duty_channel;
}
```

3．PWM 主程序文件（main.c）

```
//说明见工程文件夹下的 Doc 文件夹内 Readme.txt 文件
#define GLOBLE_VAR          //只需在 main.c 中定义一次，用来防止全局变量的重复定义

#include "includes.h"

int main(void)
 {
    //1. 声明主函数使用的变量
    uint_32 mRuncount;              //主循环计数器
    float duty=0;                   //PWM 占空比,范围 0～100，对应占空比 0%～100%
    uint_32 k;
    //2. 关总中断
    DISABLE_INTERRUPTS;             //关总中断
```

```
    //3. 初始化外设模块
    uart_init (UART_Debug,9600);        //初始化"调试串口"
    uart_init (UART_0,9600);
    uart_init (UART_1,9600);
    uart_init (UART_2,9600);
    light_init(LIGHT_0, LIGHT_OFF);     //初始化灯 0
    light_init(LIGHT_1, LIGHT_OFF);     //初始化灯 1
    light_init(LIGHT_2, LIGHT_OFF);     //初始化灯 2
    light_init(LIGHT_3, LIGHT_OFF);     //初始化灯 3
    //初始化 FTM2 模块的 PWM 功能，使用通道 0,对应 PTC0
    FTM_PWM_init(FTM_2,CHANNEL0,1,0);
    printf("Hello PWM!\r\n");           //串口发送初始化提示
    //4. main 中使用的局部变量初始化
    mRuncount = 0;                      //主循环计数器
    k = 0;
    //5. 全局变量初始化
    //6. 使能模块中断
    uart_enable_re_int(UART_Debug);     //使能 UART_Debug 的接收中断
    uart_enable_re_int(UART_0);
    uart_enable_re_int(UART_1);
    uart_enable_re_int(UART_2);
    //7. 开总中断
    ENABLE_INTERRUPTS;   //开总中断

    //进入主循环
    //=========================主循环开始=================================
    for (;;)
    {
        //---------------------------运行指示灯闪烁-------------------------
        mRuncount++;                            //主循环次数计数器+1
        if (mRuncount >= RUN_COUNTER_MAX)   //主循环次数计数器大于设定的宏常数
        {
            mRuncount = 0;                      //主循环次数计数器清零
            light_change(LIGHT_1);              //灯 1（LIGHT_1）状态变化
            light_change(LIGHT_2);              //灯 2（LIGHT_2）状态变化
            light_change(LIGHT_3);              //灯 3（LIGHT_3）状态变化
        }
        //输出占空比逐渐变化的 PWM 波
        k++;
        if(k>=300000)
        {
            duty=duty+5;
            if(duty>100)
                duty=0;
            else FTM_PWM_set_duty(FTM_2,CHANNEL0,duty);//逐渐改变小灯的亮度
            k=0;
        }
```

```
        //----------------------------以下加入用户程序----------------------
    }  //主循环 end_for

    //主循环结束=================================================================
    return 0;
}
```

4．PWM 测试工程

测试工程功能概述如下。

（1）串口通信格式：波特率 9600，1 位停止位，无校验。

（2）上电或按复位按钮时，调试串口输出“Hello PWM!”。

（3）主循环中，改变 LIGHT_1～LIGHT_3 小灯状态（三盏灯同时闪烁），通过 FTM 模块通道 0（通道 0 对应引脚 PTC0）使得占空比从 0 逐渐变大到 100，再从 0 逐渐变小到 0，如此反复，因此 LIGHT_0 会逐渐变暗直到熄灭，再逐渐变亮，循环执行。

（4）PC 向 MCU 发送数据时，MCU 进入串口接收中断，将接收的一个字节直接回发。三个串口具有相同功能。

PWM 构件的测试工程位于网上光盘中的“..\program\CH08-KEA128-FTM_PWM”文件夹。

8.4　FTM 模块的输出比较功能

8.4.1　输出比较的基本知识

1．输出比较的方法

输出比较是用程序的方法在规定的时刻输出需要的电平，实现对外部电路的控制。当定时器的某一通道用作输出比较功能时，通道寄存器的值和计数寄存器的值每隔 4 个总线周期比较一次。当两个值相等时，输出比较模块置定时器通道状态和控制寄存器的 CHxF 位为 1，并且在该通道的引脚上输出预先规定的电平。如果输出比较中断允许，还会产生一个中断。

对比使用延时来得到所需输出电平的方法，使用输出比较的优势在于可以得到非常精确的输出时间间隔。硬件的比较功能不受其他中断的影响，而且对用户程序没有额外的负担。

输出比较最简单、最常用的功能就是产生一定间隔的脉冲。典型的应用实例就是实现软件的串行通信。用输入捕捉作为数据输入，而用输出比较作为数据输出。首先根据通信的波特率向通道寄存器写入延时的值，根据待传的数据位确定有效输出电平的高低。在输出比较中断处理程序中，重新更改通道寄存器的值，并根据下一位数据改写有效输出电平控制位。

2．输出比较的过程

如图 8-4 所示，周期由预置寄存器决定，而脉宽由通道寄存器决定。当计数器的值与通道寄存器的值相等时，就发生了预先设定的变化，图中由高电平翻转到低电平。这种变化由通道控制状态寄存器中的 MSnB、MSnA、ELSnB 和 ELSnA 决定，具体参见前节表 8-2 的内

容。通常我们需要置通道控制和状态寄存器的 D0 位，这样一个周期结束的时候就会发生自动的电平翻转。

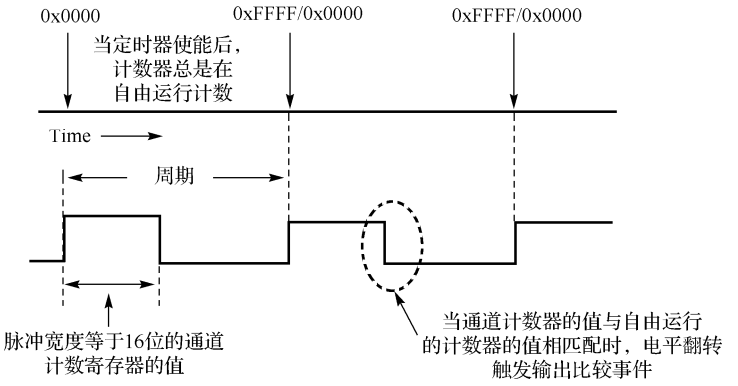


图 8-4 输出比较过程

8.4.2 输出比较构件与测试实例

以下给出了一个简单的输出比较测试实例，该测试实例将 PTC0 配置为输出比较功能，输出比较的周期是 100 ms，电平翻转时间是每个周期开始的 50 ms，并触发输出比较中断，执行相应的中断服务例程，中断服务例程中通过串口打印相关信息。

按照构件的思想，可将它们封装成独立的功能函数，初始化完成对输出比较模块的工作属性的设定。对输出比较进行编程，实际上已经涉及对硬件底层寄存器的直接操作，因此可以将初始化、启动和关闭等基本操作所对应的功能函数共同定义在命名为 outcomp.c 的文件中，并按照相对严格的构件设计原则对其进行封装，同时配以命名为 outcomp.h 的头文件，用来定义模块的基本信息和对外接口。将中断函数定义在名为 isr.c 的文件中。

1．输出比较驱动构件头文件（outcomp.h）

```
//============================================================================
//文件名称：outcomp.h
//功能概要：输出比较底层驱动构件源文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-05-11  V1.0
//芯片类型：KEA128
//============================================================================
#include "common.h"             //包含公共要素头文件

#ifndef FTM_H_
#define FTM_H_

//FTM 模块号
#define FTM_0 0
#define FTM_1 1
#define FTM_2 2
```

```
#define FTM0_irq_no 17
#define FTM1_irq_no 18
#define FTM2_irq_no 19

//通道号，FTM0 和 FTM1：通道 0～通道 1，FTM2：通道 0～通道 5
#define CHANNEL0 0
#define CHANNEL1 1
#define CHANNEL2 2
#define CHANNEL3 3
#define CHANNEL4 4
#define CHANNEL5 5
//------------------------------FTM0 通道和引脚选择-------------------------
//注：通过展开以下宏定义并修改宏定义值可选择通道号和引脚

//FTM0 通道 0 引脚组：1=PTA0 脚，2=PTB2 脚
//#define FTM0_CHANNEL0_GROUP 1    //SD-FSL-KL25-EVB 板上使用 PTA14~15 脚

//FTM0 通道 1 引脚组：1=PTA1 脚，2=PTB3 脚
//#define FTM0_CHANNEL1_GROUP 1
//---------------------------------------------------------------------------


//------------------------------FTM1 通道和引脚选择-------------------------
//注：通过展开以下宏定义并修改宏定义值可选择通道号和引脚

//FTM1 通道 0 引脚组：1=PTC4 脚，2=PTH2 脚
#define FTM1_CHANNEL0_GROUP 2  //KEA128 板上使用 PTH2 脚

//FTM1 通道 1 引脚组：1=PTC5 脚，2=PTE7 脚
//#define FTM1_CHANNEL1_GROUP 1
//---------------------------------------------------------------------------


//------------------------------FTM2 通道和引脚选择-------------------------
//注：通过展开以下宏定义并修改宏定义值可选择通道号和引脚

//FTM2 通道 0 引脚组：1=PTC0 脚，2=PTH0 脚，3=PTF0 脚
//#define FTM2_CHANNEL0_GROUP 1

//FTM2 通道 1 引脚组：1=PTC1 脚，2=PTH1 脚，3=PTF1 脚
//#define FTM2_CHANNEL1_GROUP 1

//FTM2 通道 2 引脚组：1=PTC2 脚，2=PTD0 脚，3=PTG4 脚
//#define FTM2_CHANNEL2_GROUP 1

//FTM2 通道 3 引脚组：1=PTC3 脚，2=PTD1 脚，3=PTG5 脚
//#define FTM2_CHANNEL3_GROUP 1
```

```
//FTM2 通道 4 引脚组：1=PTB4 脚，2=PTG6 脚
//#define FTM2_CHANNEL4_GROUP 1

//FTM2 通道 5 引脚组：1=PTB5 脚，2=PTG7 脚
//#define FTM2_CHANNEL5_GROUP  1     //KEA128 板上使用 PTB5 脚
//-------------------------------------------------------------------------

//=========================================================================
//函数名称：FTM_OutCampareInit
//函数返回：无
//参数说明：ftmNo:FTM 模块号，FTM_0、FTM_1、FTM_2
//         channel:选择通道号，FTM0 和 FTM1：通道 0、通道 1
//                             FTM2:通道 0、1、2、3、4、5
//功能概要：初始化输出比较模块，开启 channel 中断
//=========================================================================
void FTM_OutCompareInit(uint_8 ftmNo,uint_8 channel);

//=========================================================================
//函数名称：enable_ftm_int
//函数返回：无
//参数说明：FTMChannel:FTM 通道号
//功能概要：开 FTM 中断
//=========================================================================
void enable_ftm_int( uint_8 FTMChannel);

//=========================================================================
//函数名称：disable_ftm_int
//函数返回：无
//参数说明：FTMChannel:FTM 通道号
//功能概要：关 FTM 中断
//=========================================================================
void disable_ftm_int(uint_8 FTMChannel);

#endif // FTM_H
```

2．输出比较驱动构件源文件（outcomp.c）

```
//=========================================================================
//文件名称：outcomp.c
//功能概要：输出比较底层驱动构件源文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-05-11  V1.0
//芯片类型：KEA128
//=========================================================================
#include "outcomp.h"
```

```
//FTM0、FTM1、FTM2 地址映射
FTM_MemMapPtr FTM_ARR[]={FTM0_BASE_PTR,FTM1_BASE_PTR,FTM2_BASE_PTR};

//=============================================================================
//函数名称：FTM_OutCampareInit
//函数返回：无
//参数说明：ftmNo:FTM 模块号，FTM_0、FTM_1、FTM_2
//         channel:选择通道号，FTM0 和 FTM1：通道 0、通道 1
//                             FTM2:通道 0、1、2、3、4、5
//功能概要：初始化输出比较模块，开启 channel 中断
//=============================================================================
void FTM_OutCompareInit(uint_8 ftmNo,uint_8 channel)
{
    //防止越界
    if(ftmNo>2) ftmNo=2;
    switch(ftmNo)
    {
        case FTM_0:
            #if(FTM0_CHANNEL0_GROUP==1)
                //复用引脚 PTA0 为 FTM0 通道 0 功能
                SIM_PINSEL0 &= ~SIM_PINSEL_FTM0PS0_MASK;
            #endif

            #if(FTM0_CHANNEL0_GROUP==2)
                //复用引脚 PTB2 为 FTM0 通道 0 功能
                SIM_PINSEL0 |= SIM_PINSEL_FTM0PS0_MASK;
            #endif

            #if(FTM0_CHANNEL1_GROUP==1)
                //复用引脚 PTA1 为 FTM0 通道 1 功能
                SIM_PINSEL0 &= ~SIM_PINSEL_FTM0PS1_MASK;
            #endif

            #if(FTM0_CHANNEL1_GROUP==2)
                //复用引脚 PTB3 为 FTM0 通道 1 功能
                SIM_PINSEL0 |= SIM_PINSEL_FTM0PS1_MASK;
            #endif

            //使能 FTM0 的总线时钟
            SIM_SCGC |= SIM_SCGC_FTM0_MASK;
            break;

        case FTM_1:
            #if(FTM1_CHANNEL0_GROUP==1)
                //复用引脚 PTC4 为 FTM1 通道 0 功能
                SIM_PINSEL0 &= ~SIM_PINSEL_FTM1PS0_MASK;
            #endif
```

```
            #if(FTM1_CHANNEL0_GROUP==2)
                //复用引脚 PTH2 为 FTM1 通道 0 功能
                SIM_PINSEL0 |= SIM_PINSEL_FTM1PS0_MASK;
            #endif

            #if(FTM1_CHANNEL1_GROUP==1)
                //复用引脚 PTC5 为 FTM1 通道 1 功能
                SIM_PINSEL0 &= ~SIM_PINSEL_FTM1PS1_MASK;
            #endif

            #if(FTM1_CHANNEL1_GROUP==2)
                //复用引脚 PTE7 为 FTM1 通道 1 功能
                SIM_PINSEL0 |= SIM_PINSEL_FTM1PS1_MASK;
            #endif

            //使能 FTM1 的总线时钟
            SIM_SCGC |= SIM_SCGC_FTM1_MASK;
            break;

        case FTM_2:
            #if(FTM2_CHANNEL0_GROUP==1)
                //复用引脚 PTC0 为 FTM2 通道 0 功能
                SIM_PINSEL1 &= ~SIM_PINSEL1_FTM2PS0_MASK;
            #endif

            #if(FTM2_CHANNEL0_GROUP==2)
                //复用引脚 PTH0 为 FTM2 通道 0 功能
                SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS0(1);
            #endif

            #if(FTM2_CHANNEL0_GROUP==3)
                //复用引脚 PTF0 为 FTM2 通道 0 功能
                SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS0(2);
            #endif

            #if(FTM2_CHANNEL1_GROUP==1)
                //复用引脚 PTC1 为 FTM2 通道 1 功能
                SIM_PINSEL1 &= ~SIM_PINSEL1_FTM2PS1_MASK;
            #endif

            #if(FTM2_CHANNEL1_GROUP==2)
                //复用引脚 PTH1 为 FTM2 通道 1 功能
                SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS1(1);
                #endif

            #if(FTM2_CHANNEL1_GROUP==3)
```

```
    //复用引脚 PTF1 为 FTM2 通道 1 功能
    SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS1(2);
#endif

#if(FTM2_CHANNEL2_GROUP==1)
    //复用引脚 PTC2 为 FTM2 通道 2 功能
    SIM_PINSEL1 &= ~SIM_PINSEL1_FTM2PS2_MASK;
#endif

#if(FTM2_CHANNEL2_GROUP==2)
    //复用引脚 PTD0 为 FTM2 通道 2 功能
    SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS2(1);
#endif

#if(FTM2_CHANNEL2_GROUP==3)
     //复用引脚 PTG4 为 FTM2 通道 2 功能
     SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS2(2);
#endif

#if(FTM2_CHANNEL3_GROUP==1)
    //复用引脚 PTC3 为 FTM2 通道 3 功能
    SIM_PINSEL1 &= ~SIM_PINSEL1_FTM2PS3_MASK;
#endif

#if(FTM2_CHANNEL3_GROUP==2)
    //复用引脚 PTD1 为 FTM2 通道 3 功能
    SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS3(1);
#endif

#if(FTM2_CHANNEL3_GROUP==3)
    //复用引脚 PTG5 为 FTM2 通道 3 功能
    SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS3(2);
#endif

#if(FTM2_CHANNEL4_GROUP==1)
    //复用引脚 PTB4 为 FTM2 通道 4 功能
    SIM_PINSEL1 &= ~SIM_PINSEL1_FTM2PS4_MASK;
#endif

#if(FTM2_CHANNEL4_GROUP==2)
    //复用引脚 PTG6 为 FTM2 通道 4 功能
    SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS4_MASK;
#endif

#if(FTM2_CHANNEL5_GROUP==1)
    //复用引脚 PTB5 为 FTM2 通道 5 功能
    SIM_PINSEL1 &= ~SIM_PINSEL1_FTM2PS5_MASK;
```

```
            #endif

            #if(FTM2_CHANNEL5_GROUP==2)
                //复用引脚 PTG7 为 FTM2 通道 5 功能
                SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS5_MASK;
            #endif

            //使能 FTM2 的总线时钟
            SIM_SCGC |= SIM_SCGC_FTM2_MASK;
            break;
    }

    //总线时钟频率 24 MHz,分频因子 128,分频后为 187.5 kHz
    FTM_SC_REG(FTM_ARR[ftmNo])=FTM_SC_PS(7);
    //计数器清 0
    FTM_CNT_REG(FTM_ARR[ftmNo])=0;
    //周期=(MOD-CNTIN+1)*时钟周期=100 ms
    FTM_MOD_REG(FTM_ARR[ftmNo])=18750;
    //使能 channel 中断
    FTM_CnSC_REG(FTM_ARR[ftmNo],channel) |= FTM_CnSC_CHIE_MASK;
    // 配置成输出比较模式
    FTM_CnSC_REG(FTM_ARR[ftmNo],channel) &=~FTM_CnSC_MSB_MASK;
    FTM_CnSC_REG(FTM_ARR[ftmNo],channel) |=FTM_CnSC_MSA_MASK;
    FTM_CnSC_REG(FTM_ARR[ftmNo],channel) |=FTM_CnSC_ELSA_MASK;
    //设置通道值寄存器
    FTM_CnV_REG(FTM_ARR[ftmNo],channel)=9375;
    //使用系统时钟
    FTM_SC_REG(FTM_ARR[ftmNo]) |= FTM_SC_CLKS(1);
}

//=========================================================================
//函数名称：enable_ftm_int
//函数返回：无
//参数说明：FTMChannel:FTM 通道号
//功能概要：开 FTM 中断
//=========================================================================
void enable_ftm_int( uint_8 FTMChannel)
{
//开 FTM 中断
    NVIC_EnableIRQ(FTM1_irq_no + FTMChannel);
}

//=========================================================================
//函数名称：disable_ftm_int
//函数返回：无
//参数说明：FTMChannel:FTM 通道号
//功能概要：关 FTM 中断
```

```
//=========================================================================
void disable_ftm_int(uint_8 FTMChannel)
{
    //关 FTM 中断
    NVIC_DisableIRQ(FTM1_irq_no + FTMChannel);
}
```

3．输出比较主程序文件（main.c）

```
//说明见工程文件夹下的 Doc 文件夹内 Readme.txt 文件
#define GLOBLE_VAR  //只需在 main.c 中定义一次，用来防止全局变量的重复定义

#include "includes.h"

int main(void)
 {
    //1. 声明主函数使用的变量
    uint_32 mRuncount;
    //2. 关总中断
    DISABLE_INTERRUPTS;                         //关总中断
    //3. 初始化外设模块
    uart_init (UART_Debug,9600);                //初始化"调试串口"
    uart_init (UART_0,9600);
    uart_init (UART_1,9600);
    uart_init (UART_2,9600);
    light_init(LIGHT_0, LIGHT_OFF);             //初始化灯 0
    light_init(LIGHT_1, LIGHT_OFF);             //初始化灯 1
    light_init(LIGHT_2, LIGHT_OFF);             //初始化灯 2
    light_init(LIGHT_3, LIGHT_OFF);             //初始化灯 3
    FTM_OutCompareInit(FTM_1,CHANNEL0);  //初始化 FTM1 模块的输出比较功能，使用通道 0
    printf("Hello FTM_OutCompare! \r\n");  //串口发送初始化提示
    //4. main 中使用的局部变量初始化
    mRuncount = 0;
    //5. 全局变量初始化
    //6. 使能模块中断
    uart_enable_re_int(UART_Debug);     //使能 UART_Debug 的接收中断
    uart_enable_re_int(UART_0);
    uart_enable_re_int(UART_1);
    uart_enable_re_int(UART_2);
    enable_ftm_int(CHANNEL0);
    //7. 开总中断
    ENABLE_INTERRUPTS;   //开总中断

    //进入主循环
    //主循环开始=============================================================
    for (;;)
    {
```

```
            mRuncount++;
            //运行指示灯闪烁-----------------------------------------------
            if (mRuncount >= RUN_COUNTER_MAX)   //输入捕捉计数器大于设定的宏常数
            {
                mRuncount = 0;                  //主循环计数器清 0
                light_change(LIGHT_2);          //灯 2（LIGHT_2）状态变化
                light_change(LIGHT_3);          //灯 3（LIGHT_3）状态变化
            }
            //以下加入用户程序-----------------------------------------------
        } //主循环 end_for

        //主循环结束=============================================================
        return 0;
    }
```

4．输出比较测试工程

测试工程功能概述如下。

（1）串口通信格式：波特率 9600，1 位停止位，无校验。

（2）上电或按复位按钮时，调试串口输出“Hello FTM_OutCompare!”。

（3）主循环中，改变 LIGHT_2～LIGHT_3 小灯状态（闪烁）。

（4）FTM 每隔 100 ms 执行一次输出比较中断服务例程，调试串口输出“FTM_OutCompare”。

（5）PC 向 MCU 发送数据时，MCU 进入串口接收中断，将接收的一个字节直接回发。三个串口具有相同功能。

输出比较构件的测试工程位于网上光盘中的“..\program\CH08-KEA128-FTM_OutCompare”文件夹。

8.5 FTM 模块的输入捕捉功能

8.5.1 输入捕捉基本含义

输入捕捉功能用来监测外部的事件和输入信号。当外部事件发生或输入信号发生变化时，在指定的输入捕捉引脚上发生一个指定的沿跳变（可以设定是上升沿还是下降沿）。定时器捕捉到特定的沿跳变后，把计数寄存器当前的值锁存到通道寄存器。如果在输入捕捉控制寄存器中设定允许输入捕捉中断，系统会产生一次输入捕捉中断，利用中断处理程序可以得到事件发生时刻或信号发生变化的时刻。

通过记录输入信号的连续的沿跳变，就可以用软件算出输入信号的周期和脉宽。例如，为了测量周期，只要捕捉到两个相邻的上升沿或下降沿的时间，两者相减就可以得到周期；为了测量脉宽只要记录相邻的两个不同极性的沿变化的时间。当测量的脉宽值小于定时器的

溢出周期时，只要将两次的值直接相减（看成无符号数）。如果测量值大于定时器的溢出周期，那么在两次输入捕捉中断之间就会发生定时器计数的溢出翻转，这时直接将两个数相减就没有意义，需要考虑到定时器的溢出次数。

图 8-5 表示输入捕捉引脚的电平变化。假设触发方式是跳变沿触发，这由通道控制状态寄存器中的 MSB、MSA、ELSB 和 ELSA 决定。图中时刻 1 将计数器的值锁存在通道寄存器中，在输入捕捉中断中，把它另存到一个内存单元以防下次将内容覆盖；图中时刻 2 会再次进入中断，这次将通道寄存器的值和内存单元的值相减就得到了为低电平的时间。



图 8-5　输入捕捉过程

8.5.2　输入捕捉中断构件与测试实例

测试实例实现利用输入捕捉的引脚去捕捉输出比较引脚产生的沿跳变，并产生相应的中断，因此该输入捕捉中断测试实例是在输出比较的基础上进行的。该测试实例将 PTH2 配置为输出比较，将 PTC0 配置成输入捕捉，测试时需使用导线将这两个引脚相连，从而在 PTC0 引脚捕捉到 PTH2 引脚产生的上升沿，并执行相应的中断服务例程。

按照构件的思想，可将它们封装成独立的功能函数，初始化完成对输入捕捉模块的工作属性的设定。对输入捕捉进行编程，实际上已经涉及对硬件底层寄存器的直接操作，因此可以将初始化、启动和关闭等基本操作所对应的功能函数共同定义在命名为 incap.c 的文中，并按照相对严格的构件设计原则对其进行封装，同时配以命名为 incap.h 的头文件，用来定义模块的基本信息和对外接口。将中断函数定义在名为 isr.c 的文件中。

1．输入捕捉驱动构件头文件（incap.h）

```
//=============================================================================
//文件名称：incap.h
//功能概要：输出捕捉底层驱动构件源文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-05-11  V1.0
//芯片类型：KEA128
//=============================================================================
#include "common.h"             //包含公共要素头文件

#ifndef INCAP_H_
#define INCAP_H_

//FTM 模块号
#define FTM_0 0
#define FTM_1 1
#define FTM_2 2

#define FTM0_irq_no 17
```

```
#define FTM1_irq_no 18
#define FTM2_irq_no 19

//通道号，FTM0 和 FTM1：通道 0～通道 1，FTM2：通道 0～通道 5
#define CHANNEL0 0
#define CHANNEL1 1
#define CHANNEL2 2
#define CHANNEL3 3
#define CHANNEL4 4
#define CHANNEL5 5

//----------------------------FTM0 通道和引脚选择----------------------------
//注：通过展开以下宏定义并修改宏定义值可选择通道号和引脚

//FTM0 通道 0 引脚组：1=PTA0 脚，2=PTB2 脚
//#define FTM0_CHANNEL0_GROUP 1    //SD-FSL-KL25-EVB 板上使用 PTA14~15 脚

//FTM0 通道 1 引脚组：1=PTA1 脚，2=PTB3 脚
//#define FTM0_CHANNEL1_GROUP 1
//--------------------------------------------------------------------------


//-----------------------------FTM1 通道和引脚选择-------------------------
//注：通过展开以下宏定义并修改宏定义值可选择通道号和引脚

//FTM1 通道 0 引脚组：1=PTC4 脚，2=PTH2 脚
//#define FTM1_CHANNEL0_GROUP 2

//FTM1 通道 1 引脚组：1=PTC5 脚，2=PTE7 脚
//#define FTM1_CHANNEL1_GROUP 1
//--------------------------------------------------------------------------


//-----------------------------FTM2 通道和引脚选择-------------------------
//注：通过展开以下宏定义并修改宏定义值可选择通道号和引脚

//FTM2 通道 0 引脚组：1=PTC0 脚，2=PTH0 脚，3=PTF0 脚
#define FTM2_CHANNEL0_GROUP 1  //KEA128 板上使用 PTC0 脚

//FTM2 通道 1 引脚组：1=PTC1 脚，2=PTH1 脚，3=PTF1 脚
//#define FTM2_CHANNEL1_GROUP 1

//FTM2 通道 2 引脚组：1=PTC2 脚，2=PTD0 脚，3=PTG4 脚
//#define FTM2_CHANNEL2_GROUP 1

//FTM2 通道 3 引脚组：1=PTC3 脚，2=PTD1 脚，3=PTG5 脚
//#define FTM2_CHANNEL3_GROUP 1
```

```
//FTM2 通道 4 引脚组：1=PTB4 脚，2=PTG6 脚
//#define FTM2_CHANNEL4_GROUP 1

//FTM2 通道 5 引脚组：1=PTB5 脚，2=PTG7 脚
//#define FTM2_CHANNEL5_GROUP  1     //KEA128 板上使用 PTB5 脚
//-------------------------------------------------------------------------

//=========================================================================
//函数名称：FTM_INCaptureInit
//函数返回：无
//参数说明：ftmNo:FTM 模块号，FTM_0、FTM_1、FTM_2
//         channel:选择通道号，FTM0 和 FTM1：通道 0、通道 1
//                             FTM2:通道 0、1、2、3、4、5
//功能概要：初始化输入捕捉模块
//=========================================================================
void FTM_INCaptureInit(uint_8 ftmNo,uint_8 channel);

//=========================================================================
//函数名称：enable_ftm_int
//函数返回：无
//参数说明：FTMChannel:FTM 通道号
//功能概要：开 FTM 中断
//=========================================================================
void enable_ftm_interrupt( uint_8 FTMChannel);

//=========================================================================
//函数名称：disable_ftm_int
//函数返回：无
//参数说明：FTMChannel:FTM 通道号
//功能概要：关 FTM 中断
//=========================================================================
void disable_ftm_interrupt(uint_8 FTMChannel);


#endif /* OUTCOMP_H_ */
```

2．输入捕捉驱动构件源文件（incap.c）

```
//=========================================================================
//文件名称：incap.c
//功能概要：输入捕捉底层驱动构件源文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-05-11  V1.0
//芯片类型：KEA128
//=========================================================================
#include "incap.h"
```

```
//FTM0、FTM1、FTM2 地址映射
 FTM_MemMapPtr FTM_ARR2[]={FTM0_BASE_PTR,FTM1_BASE_PTR,FTM2_BASE_PTR};

//=============================================================================
//函数名称：FTM_INCaptureInit
//参数说明：ftmNo:FTM 模块号，FTM_0、FTM_1、FTM_2
//         channel:选择通道号，FTM0 和 FTM1：通道 0、通道 1
//                             FTM2:通道 0、1、2、3、4、5
//功能概要：初始化输入捕捉模块
//=============================================================================
void FTM_INCaptureInit(uint_8 ftmNo,uint_8 channel)
{
    //防止越界
    if(ftmNo>2) ftmNo=2;
    switch(ftmNo)
    {
        case FTM_0:
            #if(FTM0_CHANNEL0_GROUP==1)
                //复用引脚 PTA0 为 FTM0 通道 0 功能
                SIM_PINSEL0 &= ~SIM_PINSEL_FTM0PS0_MASK;
            #endif

            #if(FTM0_CHANNEL0_GROUP==2)
                //复用引脚 PTB2 为 FTM0 通道 0 功能
                SIM_PINSEL0 |= SIM_PINSEL_FTM0PS0_MASK;
            #endif

            #if(FTM0_CHANNEL1_GROUP==1)
                //复用引脚 PTA1 为 FTM0 通道 1 功能
                SIM_PINSEL0 &= ~SIM_PINSEL_FTM0PS1_MASK;
            #endif

            #if(FTM0_CHANNEL1_GROUP==2)
                //复用引脚 PTB3 为 FTM0 通道 1 功能
                SIM_PINSEL0 |= SIM_PINSEL_FTM0PS1_MASK;
            #endif

            //使能 FTM0 的总线时钟
            SIM_SCGC |= SIM_SCGC_FTM0_MASK;
            break;

        case FTM_1:
            #if(FTM1_CHANNEL0_GROUP==1)
                //复用引脚 PTC4 为 FTM1 通道 0 功能
                SIM_PINSEL0 &= ~SIM_PINSEL_FTM1PS0_MASK;
            #endif
```

```
        #if(FTM1_CHANNEL0_GROUP==2)
            //复用引脚 PTH2 为 FTM1 通道 0 功能
            SIM_PINSEL0 |= SIM_PINSEL_FTM1PS0_MASK;
        #endif

        #if(FTM1_CHANNEL1_GROUP==1)
            //复用引脚 PTC5 为 FTM1 通道 1 功能
            SIM_PINSEL0 &= ~SIM_PINSEL_FTM1PS1_MASK;
        #endif

        #if(FTM1_CHANNEL1_GROUP==2)
            //复用引脚 PTE7 为 FTM1 通道 1 功能
            SIM_PINSEL0 |= SIM_PINSEL_FTM1PS1_MASK;
        #endif

        //使能 FTM1 的总线时钟
        SIM_SCGC |= SIM_SCGC_FTM1_MASK;
        break;

    case FTM_2:
        #if(FTM2_CHANNEL0_GROUP==1)
            //复用引脚 PTC0 为 FTM2 通道 0 功能
            SIM_PINSEL1 &= ~SIM_PINSEL1_FTM2PS0_MASK;
        #endif

        #if(FTM2_CHANNEL0_GROUP==2)
            //复用引脚 PTH0 为 FTM2 通道 0 功能
            SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS0(1);
        #endif

        #if(FTM2_CHANNEL0_GROUP==3)
            //复用引脚 PTF0 为 FTM2 通道 0 功能
            SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS0(2);
        #endif

        #if(FTM2_CHANNEL1_GROUP==1)
            //复用引脚 PTC1 为 FTM2 通道 1 功能
            SIM_PINSEL1 &= ~SIM_PINSEL1_FTM2PS1_MASK;
        #endif

        #if(FTM2_CHANNEL1_GROUP==2)
            //复用引脚 PTH1 为 FTM2 通道 1 功能
            SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS1(1);
        #endif

        #if(FTM2_CHANNEL1_GROUP==3)
            //复用引脚 PTF1 为 FTM2 通道 1 功能
```

```
    SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS1(2);
#endif

#if(FTM2_CHANNEL2_GROUP==1)
    //复用引脚 PTC2 为 FTM2 通道 2 功能
    SIM_PINSEL1 &= ~SIM_PINSEL1_FTM2PS2_MASK;
#endif

#if(FTM2_CHANNEL2_GROUP==2)
    //复用引脚 PTD0 为 FTM2 通道 2 功能
    SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS2(1);
#endif

#if(FTM2_CHANNEL2_GROUP==3)
     //复用引脚 PTG4 为 FTM2 通道 2 功能
     SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS2(2);
#endif

#if(FTM2_CHANNEL3_GROUP==1)
    //复用引脚 PTC3 为 FTM2 通道 3 功能
    SIM_PINSEL1 &= ~SIM_PINSEL1_FTM2PS3_MASK;
#endif

#if(FTM2_CHANNEL3_GROUP==2)
    //复用引脚 PTD1 为 FTM2 通道 3 功能
    SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS3(1);
#endif

#if(FTM2_CHANNEL3_GROUP==3)
   //复用引脚 PTG5 为 FTM2 通道 3 功能
   SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS3(2);
#endif

#if(FTM2_CHANNEL4_GROUP==1)
    //复用引脚 PTB4 为 FTM2 通道 4 功能
    SIM_PINSEL1 &= ~SIM_PINSEL1_FTM2PS4_MASK;
#endif

#if(FTM2_CHANNEL4_GROUP==2)
    //复用引脚 PTG6 为 FTM2 通道 4 功能
    SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS4_MASK;
#endif

#if(FTM2_CHANNEL5_GROUP==1)
    //复用引脚 PTB5 为 FTM2 通道 5 功能
    SIM_PINSEL1 &= ~SIM_PINSEL1_FTM2PS5_MASK;
#endif
```

```
            #if(FTM2_CHANNEL5_GROUP==2)
                //复用引脚 PTG7 为 FTM2 通道 5 功能
                SIM_PINSEL1 |= SIM_PINSEL1_FTM2PS5_MASK;
            #endif

            //使能 FTM2 的总线时钟
            SIM_SCGC |= SIM_SCGC_FTM2_MASK;
            break;
    }
    //总线时钟频率 24 MHz,分频因子 128,分频后为 187.5 kHz
    FTM_SC_REG(FTM_ARR2[ftmNo]) |= FTM_SC_PS(7);
    //计数器清 0
    FTM_CNT_REG(FTM_ARR2[ftmNo])=0;
    //周期 500ms，为了保证捕捉信号时不被定时器溢出中断打断，需将 MOD 值尽可能设置大一点
    FTM_MOD_REG(FTM_ARR2[ftmNo])=93750;

    //设置为上升沿捕捉
    FTM_CnSC_REG(FTM_ARR2[ftmNo],channel) &=～FTM_CnSC_MSA_MASK;
    FTM_CnSC_REG(FTM_ARR2[ftmNo],channel) &=～FTM_CnSC_MSB_MASK;
    FTM_CnSC_REG(FTM_ARR2[ftmNo],channel) &=～FTM_CnSC_ELSB_MASK;
    FTM_CnSC_REG(FTM_ARR2[ftmNo],channel) |=FTM_CnSC_ELSA_MASK;
    //channel 中断使能
    FTM_CnSC_REG(FTM_ARR2[ftmNo],channel) |=FTM_CnSC_CHIE_MASK;

    //使用系统时钟
    FTM_SC_REG(FTM_ARR2[ftmNo]) |= FTM_SC_CLKS(1);
}

//=========================================================================
//函数名称：enable_ftm_int
//函数返回：无
//参数说明：FTMChannel:FTM 通道号
//功能概要：开 FTM 中断
//=========================================================================
void enable_ftm_interrupt( uint_8 FTMChannel)
{
    //开 FTM 中断
    NVIC_EnableIRQ(FTM2_irq_no + FTMChannel);
}

//=========================================================================
//函数名称：disable_ftm_int
//函数返回：无
//参数说明：FTMChannel:FTM 通道号
//功能概要：关 FTM 中断
//=========================================================================
```

```
void disable_ftm_interrupt(uint_8 FTMChannel)
{
    //关 FTM 中断
    NVIC_DisableIRQ(FTM2_irq_no + FTMChannel);
}
```

3．输入捕捉主程序文件（main.c）

```
//说明见工程文件夹下的 Doc 文件夹内 Readme.txt 文件
#define GLOBLE_VAR          //只需在 main.c 中定义一次，用来防止全局变量的重复定义

#include "includes.h"

int main(void)
 {
    //1. 声明主函数使用的变量
    uint_32 mRuncount;                  //主循环计数器
    //2. 关总中断
    DISABLE_INTERRUPTS;                 //关总中断
    //3. 初始化外设模块
    uart_init (UART_Debug,9600);        //初始化"调试串口"
    uart_init (UART_0,9600);
    uart_init (UART_1,9600);
    uart_init (UART_2,9600);
    light_init(LIGHT_0, LIGHT_OFF);     //初始化灯 0
    light_init(LIGHT_1, LIGHT_OFF);     //初始化灯 1
    light_init(LIGHT_2, LIGHT_OFF);     //初始化灯 2
    light_init(LIGHT_3, LIGHT_OFF);     //初始化灯 3
    FTM_OutCompareInit(FTM_1,CHANNEL0); //初始化 FTM1 模块的输出比较功能，使用通道 0
    FTM_INCaptureInit(FTM_2,CHANNEL0); //初始化 FTM2 模块的输入捕捉功能，使用通道 0
    printf("Hello FTM_InCapture!\r\n");//串口发送初始化提示
    //4. main 中使用的局部变量初始化
    mRuncount=0;
    //5. 全局变量初始化
    //6. 使能模块中断
    uart_enable_re_int(UART_Debug);     //使能 UART_Debug 的接收中断
    uart_enable_re_int(UART_0);
    uart_enable_re_int(UART_1);
    uart_enable_re_int(UART_2);
    enable_ftm_interrupt(CHANNEL0);
    //7. 开总中断
    ENABLE_INTERRUPTS;                  //开总中断

    //进入主循环
    //主循环开始============================================================
    for (;;)
    {
        mRuncount++;
```

```
        //运行指示灯闪烁-------------------------------------------------
        if (mRuncount >= RUN_COUNTER_MAX)   //输入捕捉计数器大于设定的宏常数
        {
            mRuncount = 0;                  //主循环计数器清 0
            light_change(LIGHT_2);          //灯 2（LIGHT_2）状态变化
            light_change(LIGHT_3);          //灯 3（LIGHT_3）状态变化
        }
        //以下加入用户程序-----------------------------------------------
    }  //主循环 end_for

    //主循环结束=================================================================
    return 0;
}
```

4．输入捕捉中断服务例程源文件（isr.c）

```
//输入捕捉中断服务例程
void isr_incap(void)
{
    if((FTM2_C0SC & FTM_CnSC_CHF_MASK) == FTM_CnSC_CHF_MASK)
    //捕捉到上升沿，CHF 位被置位
    {
        uint_16 ftmValue;
        ftmValue = FTM2_C0V;
        //printf("FTM_InCapture\n");
        FTM2_C0SC &= ～FTM_CnSC_CHF_MASK;        //清 CHF 位
        FTM2_CNT = 0;
        printf("Capture value: %d\n",ftmValue);
    }
}
```

5．输入捕捉测试工程

测试工程功能概述如下。

（1）串口通信格式：波特率 9600，1 位停止位，无校验。

（2）上电时，调试串口输出“Hello FTM_InCapture!”。

（3）主循环中，改变 LIGHT_2～LIGHT_3 小灯状态（闪烁）。

（4）PTC0 与 PTH2 相连后，PTC0 开始捕捉 PTH2 的上升沿，同时触发输入捕捉中断，在相应的中断服务例程中打印串口信息“Capture value:"9286"”，周期是 100 ms。

（5）PC 向 MCU 发送数据时，MCU 进入串口接收中断，将接收的一个字节直接回发。三个串口具有相同功能。

输入捕捉构件的测试工程位于网上光盘中的“..\program\CH08-KEA128-FTM_InCapture”文件夹。

第 9 章

Flash 在线编程

本章导读：本章主要介绍了 KEA128 的 FLASH 构件及使用方法，主要从使用的角度，简要介绍 Flash 编程知识要素，并给出 Flash 驱动构件头文件及使用方法。从 Flash 寄存器映射等方面介绍构件的制作方法和编程结构，给出构件封装要点和实现。最后，从保护和加密、解密两个方面，对 KEA128 FLash 的安全与保护特性进行了介绍，并给出 Flash 存储保护驱动构件的简要实现。

本章阅读资料：本章 9.2 节和 9.4 节参考自 KEA 参考手册第 18、19 章。

9.1 KEA128 芯片 Flash 驱动构件及使用方法

KEA128 的存储器模块缩写为 FTMRE（Freescale TFS Memory），其中 TFS 是指薄膜存储器（Thin Film Storage），它是一种 Flash 的制作工艺。

KEA128 芯片 Flash 模块以扇区为基本组织单位，每个扇区的大小为 512 B。在线编程时，擦除以扇区为单位进行；内建擦除与写入算法，简化了编程过程；具有保护机制以防止意外擦除或写入。KEA128 芯片 128 KB 的 Flash 地址范围是 0x0000_0000～0x0001_FFFF，分为 256 个扇区，扇区编号为 0～255，实际编程时以扇区为逻辑单位。

9.1.1 Flash 编程知识要素

从 Flash 存储器的基本特点可以看出，在 MCU 中，可以利用 Flash 存储器固化程序，这一般通过编程器来完成。Flash 存储器工作于这种情况，叫做监控模式或写入器模式，这与一般的 EPROM 装入程序的含义相似。另一方面，由于 Flash 存储器具有电可擦除功能，因此，在程序运行过程中，有可能对 Flash 存储区的数据或程序进行更新，Flash 存储器工作于这种情况，叫做用户模式或在线编程模式。但是并不是所有类型的 MCU 内 Flash 存储器都具有在线编程功能。

对 Flash 存储器的读写不同于对一般的 RAM 读写，需要专门的编程过程。Flash 编程的基本操作有两种：擦除（Erase）和写入（Program）。擦除操作的含义是将存储单元的内容由二进制的 0 变成 1，而写入操作的含义是将存储单元的某些位由二进制的 1 变成 0。Flash 在线编程的写入操作是以字为单位进行的。在执行写入操作之前，要确保写入区在上一次擦除之后没有被写入过，即写入区是空白的（各存储单元的内容均为 0xFF），所以，在写入之前一般都要先执行擦除操作。Flash 在线编程的擦除操作包括整体擦除和以 *m* 个字为单位的擦除。这 *m* 个字在不同厂商或不同系列的 MCU 中，其称呼不同，有的称为块，有的称为页，

有的称为扇区，等等，它表示在线擦除的最小度量单位。KEA128 芯片 Flash 的写操作在 Flash 块中写入至少 4 B，擦除最小单位为扇区。

在擦除/写入过程中一般需要高于电源的电压，称为编程电压 V_p。有的芯片需要外部接入编程电压 V_p，有的芯片通过内部机制解决，取决于生产厂商的技术成熟度。有的 MCU 在擦除/写入加 V_p 的过程中，会出现不稳定，需要特殊处理。特殊处理的方法，一般是把待执行的机器代码设法放到 RAM 中执行，增加了编程复杂度。随着技术的发展，一些新的 MCU 芯片已经有了新的解决方案。在 KEA128 中，在启动 Flash 命令前，只需要在杂项模块中把平台控制器（MCM_PLACR）的 ESFC 位置 1 即可。

9.1.2 KEA128 芯片 Flash 构件头文件及使用方法

头文件 flash.h 中的内容主要可以分为两个部分，一是通过宏定义，列出 Flash 所有的操作指令，方便使用；二是给出 Flash 驱动构件对上层构件或软件所提供的最主要的三个接口函数，包括初始化函数、擦除以及写入操作。以写入操作为例，将写入操作这一过程中涉及的必要信息都设置为入口参数，包括扇区号、扇区内偏移地址、写入字节数目、源数据的缓冲区首地址，在使用时无需考虑写入的底层实现方法，实现透明封装。

1．Flash 驱动构件头文件（.h 文件）

```
//=============================================================================
//文件名称：flash.h
//功能概要：Flash 底层驱动构件头文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2012-11-12   V1.0
//=============================================================================

#ifndef FLASH_H
#define FLASH_H

#include "common.h"

//Flash 控制器宏定义，内部使用
#define CCIF    (1<<7)
#define ACCERR  (1<<5)
#define FPVIOL  (1<<4)
#define MGSTAT0 (0b11<<0)

//Flash 命令宏定义，内部使用
#define ERSVRFALLBLK  0x01 //校验已擦除所有块
#define ERSVRFBLK     0x02 //校验已擦除某块
#define ERSVRFSEC     0x03 //校验已擦除 Flash 区
#define RDONCE        0x04 //一次读取
#define PGM4          0x06 //Flash 写操作
#define PGMONCE0x07        //一次写入
#define ERSALL        0x08 //擦除所有块
```

```
#define ERSBLK        0x09   //擦除 Flash 块
#define ERSSCR        0x0A   //擦除 Flash 扇区
#define USCFLS        0x0B   //不安全 Flash
#define VFYKEY        0x0C   //校验后门密钥
#define STUSERMGLVL   0x0D   //设置用户空白标准
#define STFCTMGLVL    0x0E   //设置工厂空白标准
#define CFGNVM        0x0F   //配置 NVM

#define NONE_SIZE     0
#define SIZE_2K       4
#define SIZE_4K       8
#define SIZE_8K       16
#define SIZE_16K      32
#define LAST_112K     224    //除了前 16 KB 之外的 112 KB
#define ALL_SIZE      256

//=============================================================================
//函数名称：flash_init
//函数返回：无
//参数说明：无
//功能概要：Flash 初始化
//=============================================================================
void flash_init();

//=============================================================================
//函数名称：flash_erase
//函数返回：函数执行执行状态，0=正常；非 0=异常。
//参数说明：sectorNo：扇区号（KEA128 实际使用 0～255）每个扇区 512 B
//功能概要：擦除指定 Flash 扇区
//=============================================================================
uint_8 flash_erase_sector(uint_16 sectorNo);

//=============================================================================
//函数名称：flash_write
//函数返回：函数执行状态，0=正常；非 0=异常。
//参数说明：sectNo：目标扇区号（KEA128 实际使用 0～255）每个扇区 512 B
//         offset:写入扇区内部偏移地址（0～508）
//         cnt：写入字节数目（4～512）
//         buf：源数据缓冲区首地址
//功能概要：Flash 写入操作
//=============================================================================
uint_8 flash_write(uint_16 sectNo,uint_16 offset,uint_16 cnt,uint_8 buf[]);

//=============================================================================
//函数名称：flash_secure
//函数返回：函数执行状态，0=正常；非 0=异常。
//参数说明：无
```

```
//功能概要：Flash 加密操作
//==========================================================================
uint_8 flash_secure(uint_8 key[]);

//==========================================================================
//函数名称：flash_unsecure
//函数返回：函数执行状态，0=正常；非 0=异常。
//参数说明：无
//功能概要：Flash 解密操作
//==========================================================================
uint_8 flash_unsecure();

//==========================================================================
//函数名称：flash_protect
//函数返回：函数执行状态，0=正常；非 0=异常。
//参数说明：无
//功能概要：Flash 保护操作
//==========================================================================
void flash_protect(uint_16 size);

//==========================================================================
#endif //FLASH_H__
```

2．Flash 驱动构件的使用方法

Flash 头文件中给出了 Flash 中三个最主要的基本构件函数，包括初始化函数（flash_init()）、擦除（flash_erase_sector(uint_16 sectorNo)）及写入（flash_write(uint_16 sectNo,uint_16 offset,uint_16 cnt,uint_8 buf[])），初始化函数直接调用即可，无入口参数及返回值。擦除和写入操作类似，都返回擦除/写入的结果（正常/异常），由于擦除操作对象是整个扇区，因此入口参数仅需一个扇区号。写入操作入口参数较多，除扇区号外，还有扇区内偏移地址、写入字节数目、写入数据的缓冲区首地址。

以向 14 扇区 0 字节开始的地址写入四个字节 1234 为例：

（1）首先，要进行初始化 Flash 模块。

```
flash_init();
```

（2）因为执行写入操作之前，要确保写入区在上一次擦除之后没有被写入过，即写入区是空白的（各存储单元的内容均为 0xFF），所以在写入之前一般都要先执行擦除操作，即擦除 14 扇区。

```
flash_erase_sector(14);
```

（3）通过封装好的入口参数进行传参，进行写入操作。向 10 扇区 0 字节开始的 4 B 内写入 1234。

```
flash_write(14,0,4,"1234");
```

9.2　Flash 驱动构件的制作方法

9.2.1　Flash 模块编程结构

1．Flash 索引寄存器（FTMRE_FCCOBIX）

FCCOBIX 寄存器是用来为 NVM 存储器索引 FCCOB 寄存器的。地址：4002_0000h 基地址+1h 偏移量=4002_0001h。

该寄存器 D7～D3 位保留，只读为 0。

D2～D0，CCOBIX 通用命令寄存器索引，选择 FCCOBIX 的某一字来进行读写。

2．Flash 安全寄存器（FTMRE_FSEC）

Flash 安全寄存器（FSEC）包存所有与 MCU 和 NVM 安全相关的数据位。FSCE 的所有区域均可读但不可写。但复位序列期间，FSEC 寄存器加载的是位于 Flash 配置区域的 Flash 安全字节的内容。地址：4002_0000h 基地址+2h 偏移量=4002_0002h。

D7～D6（KEYEN）—后门密钥安全使能位（只读）。该位确定启用还是不启用访问 FTMRE 模块的密钥。当 KEYEN=00，后门密钥访问禁用；当 KEYEN=01，后门密钥访问禁用；当 KEYEN=10，后门密钥访问启用；当 KEYEN=11，后门密钥访问禁用。

D5～D2（保留）。

D1～D0（SEC）—Flash 安全位（只读）。定义 MCU 的安全状态，如果 Flash 模块在不安全模式下使用后门密钥解密，SEC 区会强制设置到 10b。当 SEC =00（SEC 状态中最佳 MCU 安全状态位），MCU 处于安全状态；当 SEC =01，MCU 处于安全状态；当 SEC =10，MCU 处于不安全状态；当 SEC =11，MCU 处于安全状态。

3．Flash 时钟分频寄存器（FTMRE_FCLKDIV）

FCLKDIV 寄存器控制写入和擦除算法中的时间事件。需要注意的是，当 Flash 指令在执行时（FSTAT[CCIF]=0），禁止写入 FCLKDIV。地址：4002_0000h 基地址+3h 偏移量=4002_0003h。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	FDIVLD	FDIVLCK	FDIV					
写	—							

注：复位值全为 0。

D7（FDIVLD）—时钟分频加载（只读）。当 FDIVLD=0 时，上次重置后 FCLKDIV 未被写入；当 FDIVLD=1，上次重置后 FCLKDIV 被写入。

D6（FDIVLCK）—时钟分频锁（读/写）。FDIVLCK=0，FDIV 设置为可写入；FDIVLCK=1，FDIV 的值被锁定并且不能更改。时钟分频锁被设置为高位后，只有通过复位能清除此位并且在用户模式中恢复可写性。

D5～D0（FDIV）—时钟分频位（读/写），必须设置 FDIV[5:0]位将数据总线有效分割成 1MHz 从而在写入和擦除算法中控制时限事件。该位的值需参照基于数据总线的推荐值写入（《KEA128 用户参考手册》18.3.4.1 编写时钟分频寄存器表 18-2），一般情况下，KEA128 的总线时钟频率为 24 MHz，因此对应时钟分频寄存器的值为 0x17。

4．Flash 状态寄存器（FTMRE_FSTAT）

FSTAT 寄存器（FTMRE_FSTAT）给出 Flash 模块的状态。CCIF、ACCERR、FPVIOL 这些数据位是可读可写的，MGBUSY、MGSTAT 数据位是只读的，无符号数据位是 0，并且不可写。地址：4002_0000h 基地址+5h 偏移量=4002_0005h

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	CCIF	0	ACCERR	FPVIOL	MGBUSY	0	MGSTAT	
写		—			—			
复位	1	0						

D7（CCIF）——命令完成中断标志位（读/写）。表明 Flash 命令已经执行完成。通过写 1 清除该标志位以启动一个新的命令，在命令执行期间该标志位保持为 0，命令执行完毕或终止时由存储控制器将其置 1。CCIF=1，Flash 命令完成；CCIF=0，当前有命令在执行。

D6 保留。该段位保留，只读为 0。

D5（ACCERR）——ACCERR 标志位（读/写）。表示非法访问 Flash 资源或启动了非法命令。当 ACCERR 为 1 时，不能清除 CCIF 标志以启动另一条命令。写 1 清该标志位。ACCERR=1，检测到访问错误；ACCERR=0，未检测到访问错误。

D4（FPVIOL）——Flash 保护区非法访问标志（读/写）。FPVIOL 标志位表示试图擦除或写入保护的 Flash 区域。当 FPVIOL 为 1 时，不能清除 CCIR 标志启动命令。写 1 清该标志位。FPVIOL=1，检测到保护区非法访问错误；FPVIOL=0，未检测到保护区非法访问错误。

D3（MGBUSY）——存储控制器忙标志（只读）。反映存储控制器的活跃状态。MGBUSY =0，存储控制闲置；MGBUSY=1，存储器正忙，正在执行一个 Flash 指令（CCIF=0）。

D2 保留。该段位保留，只读为 0。

D1～D0（MGSTAT）——存储控制器指令完成状态标志（只读）。在一个 Flash 指令或者 Flash 复位序列的执行期间如果检测到一个错误，一个或者更多的 MGSTAT 标志位将会被设置。注意：如果在复位序列执行期间检测到 2 个错误，复位值就会偏离设定的值。

5．Flash 配置寄存器（FTMRE_FCNFG）

FCNFG 寄存器提供 FTMRE 模块的功能状态信息。能够使能 Flash 指令完成中断，并强制从 CPU 中读取 Flash 数组中的错误。地址：4002_0000h 基地址+7h 偏移量=4002_0007h。

D7（CCIE）——指令完成中断使能（读/写），CCIE 数据位在 FTMRE 指令完成时控制中断的产生。当 CCIE = 0 时，指令完成中断禁用；当 CCIE = 1 时，指令完成中断使能。当 FSTAT[CCIF]被置 1 时，产生一个中断请求。

D6 保留。只读，值总是为 0。

D5（ERSAREQ）——擦除所有块请求（只读），这个数据位向内存控制器发出一个执行擦除所有块的请求，解除芯片的安全状态。ERSAREQ 不是直接可写的，但是可以通过用户间接控制。MGATE 开始执行序列时，ERSAREQ 为 1。操作完成时，ERSAREQ 为 0。ERSAREQ=0，无请求或请求完成；ERSAREQ=1，发出请求：①执行擦除所有块命令；②验证擦除后状态；③在 Flash 配置区编写安全位；④通过设置 FSEC[SEC]释放 MCU 安全性。

D4～D0 保留。只读，值总是为 0。

6．通用命令参数寄存器：低（FTMRE_FCCOBLO）

FCCOB 是一个 6 个字数组，通过 CCOBIX 索引找到其在 FCCOBIX 寄存器的位置。FCCOB 允许字节宽度的读和写。

地址：4002_0000h 基地址+8h 偏移量=4002_0008h，FTMRE_FCCOBLO 是 8 位寄存器，可读可写，**D7～D0** 为通用命令参数位 7:0，是通用命令参数寄存器的低 8 位。

7．通用命令参数寄存器：高（FTMRE_FCCOBHI）

FCCOB 是一个 6 个字数组，通过 CCOBIX 索引找到其在 FCCOBIX 寄存器的位置。FCCOB 允许字节宽度的读和写。

地址：4002_0000h 基地址+9h 偏移量=002_0009h，FTMRE_FCCOBHI 是 8 位寄存器，可读可写，**D7～D0** 为通用命令参数位 15:8，是通用命令参数寄存器的高 8 位。

8．Flash 保护寄存器（FTMRE_FPROT）

FPROT 寄存器定义了 Flash 存储器模块中哪些扇区被保护以防止写入或者擦除操作。FPROT 中未限制的位在限制条件下是可写的，该限制条件为保护区的大小只能增长，也就是说当前未被保护的区域可以增加为被保护区，但当前被保护区域不能解除保护。在复位序列执行期间，FPROT 寄存器中加载是位于 Flash 存储器全局地址为 0x40D 处的 Flash 配置域中的保护内容。要改变复位序列期间 Flash 保护中加载的内容，Flash 的顶部扇区必须处于非保护状态，且 Flash 保护字节必须被重写。

在保护区区域中改变数据会导致违反保护错误，并且 FSTAT 中的 FPVIOL 寄存器将被设置。如果 Flash 扇区受保护，则无法进行包含该扇区的擦除操作。地址：4002_0000h 基地址+Bh 偏移量=4002_000Bh。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	FPOPEN	RNV6	FPHDIS	FPHS		FPLDIS	FPLS	
写		—						
复位	—							

D7（FPOPEN）—使能 Flash 保护操作位（读/写）。FPOEN 位决定是否启用保护功能，以避免对特定区域的写入和擦除操作。当 FPOPEN=0 时，FPHDIS 和 FPLDIS 位定义了未受保护的区域地址范围（通过 FPHS 和 FPLS 设定）；当 FPOPEN=1 时，FPHDIS 和 FPLDIS 位启用保护相应的 FPHS 和 FPLS 字段中的地址范围。

D6（RNV6）—保留的非易失性位（只读）。RNV 位必须保持着被擦除状态。

D5（FPHDIS）—禁止 Flash 高位地址范围保护（读/写）。FPHDIS 位决定以全局地址 0x7FFF 结尾的 Flash 存储器模块中，是否存在受保护/未保护的特定区域。当 FPHDIS=0 时，保护/未保护启用；当 FPHDIS=1 时，保护/未保护禁用。

D4～D3（FPHS）—Flash 高位保护范围（读/写）。FPHS 决定 Flash 中受保护/未保护的区域大小。只有当 FPHDIS 位为 1 时 FPHS 才能被写入。

D2（FPLDIS）—禁止 Flash 低位地址范围保护（读/写）。FPHDIS 位决定以全局地址 0x0000 开始的 Flash 存储器模块中，是否存在受保护/未保护的特定区域。当 FPLDIS=0 时，保护/未保护启用；当 FPLDIS=1 时，保护/未保护禁用。

D1～D0（FPLS）—Flash 低位保护范围（读/写）。FPLS 位决定 Flash 中受保护/未保护的区域范围。只有当 FPLDIS 位为 1 时 FPLS 才能被写入。

9．Flash 选项寄存器（FTMRE_FOPT）

FOPT 是 Flash 选项寄存器。MCU 通过检测 Flash 选项寄存器的状态来制定自己的操作。在执行复位序列期间，FOPT 寄存器从位于全局地址 0x040F 处的配置区域的非易失性字节中，按照复位状态的指示进行加载数据值。地址：4002_0000h 基地址+Fh 偏移量=4002_000Fh。

D7～D0（NV）非易失性字节。该位被称为非易失性位。在复位序列期间，FOPT 寄存器中加载的是位于 Flash 全局地址 0x040F 处 Flash 配置区中非易失性的字节。

9.2.2 Flash 驱动构件制作要点

1．系统复位后 Flash 初始化

每次系统复位，Flash 会执行一组初始化序列，为 Flash 块配置参数、FPROT 保护寄存器和 FSEC 寄存器创建初始化值。该初始化过程将数值恢复到默认值，即使在执行系统复位序列时有错误发生，Flash 模块也会处于受保护的安全状态中。如果在系统复位过程中检测到一位错误，FSTAT[MGSTAT]位都会被置位。

FSTAT[CCIF]在初始化时被清空，NVM 模块使得所有 CPU 对这些初始化的序列保持关闭。当初始化结束后，就能够读取 Flash。FSTAT[CCIF]=1 时，这组初始化序列就被标记为完成，此时可使用用户指令。FSTAT[CCIF]=0 时，不能向 FCCOBIX 或 FCCOB 寄存器执行写操作。

当有 Flash 指令在进程中时，如果此时产生复位，这个指令会被立刻终止。正在执行的写操作或者正在擦除的扇区/块的执行状态将得不到保护。

2．FTMRE 命令操作过程

Flash 指令用来改变 flash 存储器中的内容，图 9-1 流程图为 Flash 指令的写入序列，该指令操作包括 3 个步骤。

（1）配置 Flash 的写操作和擦除操作的时钟周期。

（2）使用指令写入序列，设置 Flash 的指令参数和执行。

（3）根据 MCU 的功能模式和 MCU 的安全状态执行有效的 Flash 指令。

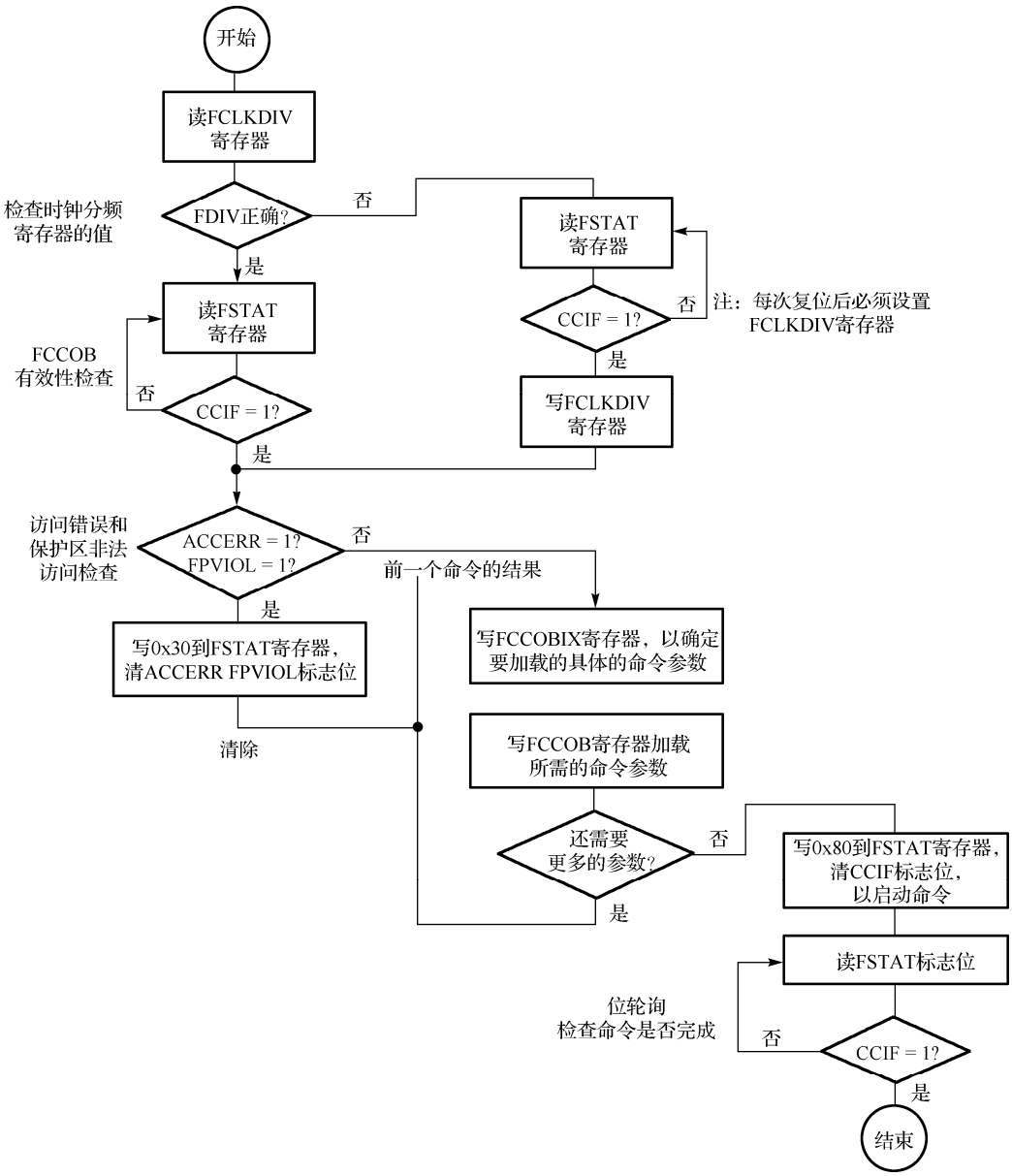


图 9-1　Flash 指令写入流程图

表 9-1 总结了有效的 Flash 指令和在 Flash 模块内指令对 Flash 块和其他资源的影响。

1）编写时钟分频寄存器（FCLKDIV）

在系统复位后执行任何 Flash 写入或擦除命令，用户必须先向 FCLKDIV 寄存器执行写操作，将总线时钟（BUSCLK）下降到 1 MHz 的目标时钟（FCLK）。在《KEA128 用户参考手册》（18.3.4.1 编写时钟分频寄存器表 18-2）中给出了基于总线时钟频率的 FCLKDIV[FDIV]推荐值，在此不再加以罗列。在样例程序中，KEA128 的总线时钟频率为 24 MHz，因此对应时钟分频寄存器的值为 0x17。

表 9-1 Flash 指令

FCMD	指 令	对 Flash 存储的功能
0x01	擦除校验所有块	校验所有已被擦除的 Flash 块
0x02	擦除校验块	校验一个已被擦除的块
0x03	擦除校验 Flash 区	检验给定的单词从提供的地址已擦除
0x04	一次读取	在非易失性寄存器中读取特定的 64 字节域，这个字节在先前已被一次写入 Flash 块中
0x06	Flash 写操作	在 Flash 块中写入至少两个字长
0x07	一次写入	编写闪存块中允许被一次写入的非易失性寄存器的一个特定的 64 位区域
0x08	擦除所有块	擦除所有 Flash 块。只有在启动命令之前设置了 FPROT[FPHDIS]、FPROT[FPLDIS]、FPROT[FPOEN]位，才可能擦除所有的 Flash 块（存储区不受保护时才能擦除）
0x09	擦除 Flash 块	擦除一个 Flash 块。只有在启动命令之前设置了 FPROT[FPHDIS]、FPROT[FPLDIS]、FPROT[FPOEN]位，才可能完整擦除 Flash 块（存储区不受保护时才能擦除）
0x0A	擦除 Flash 扇区	擦除 Flash 扇区中所有的字节
0x0B	不安全 Flash	通过擦除所有的 Flash 块和校验所有 Flash 块被擦除，支持一种释放 MCU 安全性的方法
0x0C	校验后门密钥	提供一种通过擦除所有 flash 块和验证所有 flash 块已被擦除来释放 MCU 安全性的方法
0x0D	设置用户空白标准	对所有的 Flash 块指定一个用户空白标准
0x0E	设置工厂空白标准	对所有的 Flash 块指定一个工厂空白标准
0x0F	配置 NVM	配置 NVM 参数来启用或禁用 NVM 阵列中的某些功能，允许保存在某些情况下的电流/功率

需要注意的是，当总线时钟在低于 0.8 MHz 的频率下运行时，Flash 不能够进行写入或擦除操作。FCLKDIV[FDIV]过高可能使 Flash 因为过载而损坏。而 FCLKDIV[FDIV]过低可能使得 Flash 的存储单元写入或擦除不完整。

当向 FCLKDIV 寄存器执行写操作时，FCLKDIV[FDIVLD]自动设置值。当 FCLKDIV [FDIVLD]=0 时，由于上次的复位，FCLKDIV 没有被写入，写序列期间加载的任何 Flash 写或擦除操作将不会执行并且 FSTAT[ACCERR]将会被置位。

2）写入序列命令

存储控制器使用写序列命令开始执行所有有效的 Flash 指令。需要注意在对 Flash 进行写入之前，其必须处于被擦除状态，在 Flash 中进行累积写入是不允许的。

在执行命令之前，FSTAT[ACCERR]和 FSTAT[FPVIOL]必须清 0，同时检测 FSTAT[CCIF]标志位，以决定当前写序列命令的状态。如果 FSTAT[CCIF]=0，表明之前的写命令还在执行，新的写命令序列不能开始，所有对 FCCOB 寄存器写操作都不能执行。

FCCOB 的参数域必须加载执行 Flash 指令需要的所有参数，访问 FCCOB 参数域是通过 FCCOCIX[CCOBIX]控制的。

Flash 指令模式使用有索引的 FCCOB 寄存器向存储控制器提供一个指令码和相关的参数。首先用户必须建立所有需要的 FCCOB 域，然后用户可以通过向 FSTAT[CCIF]写 1 来初始化指令的执行。这个操作清 CCIF 指令完成标志位即 FSTAT[CCIF]=0。当 FSTAT[CCIF]=0 时，直到指令完成之前（即 FSTAT[CCIF]=1），所有的 FCCOB 参数域都被锁定并且不能被用户改变。

表 9-2 给出了 Flash 指令模式的 FCCOB 参数域的通用格式。当存储控制器将 FSTAT[CCIF]标志位返回 1 后，就能够读取返回值。如向未生效的参数域执行写操作，例如，FCCOBIX [CCOBIX]=110b 和 FCCOBIX[CCOBIX]=111b，这些值将不会被执行，并返回 0x0000。

表 9-2　FCCOB-Flash 指令模式的典型使用

CCOBIX[2:0]	Byte	Flash 指令模式中的参数域	CCOBIX[2:0]	Byte	Flash 指令模式中的参数域
000	HI	FCDM[7:0]定义闪存指令	011	HI	数据 1[15:8]
	LO	全局地址[23:16]		LO	数据 1[7:0]
001	HI	全局地址[15:8]	100	HI	数据 2[15:8]
	LO	全局地址[7:0]		LO	数据 2[7:0]
010	HI	数据 0[15:8]	101	HI	数据 3[15:8]
	LO	数据 0[7:0]		LO	数据 3[7:0]

当 FSTAT[CCIF]=1，即清指令的完成标志位置 1 时，FCCOB 参数域的内容会被转移到存储控制器中。

3）扇区擦除命令

擦除 Flash 扇区操作将会擦除 Flash 扇区中的所有地址。表 9-3 为通用的 Flash 指令格式。FCCOB 数组中第一个字的高字节为指令模式，后面是 Flash 指令特定的参数。每个指令的对 FCCOB 设置需符合表 9-1 中罗列的具体指令对该字节进行设置。因此，在擦除扇区指令中，需要对 FCCOB 进行如表 9-3 所示的设置。

表 9-3　擦除 Flash 扇区指令的 FCCOB 要求

CCOBIX[2:0]	FCCOB 参数	
000	0x0A	全局地址[23:16]用来定义需要被擦除的闪存块
001	擦除 Flash 扇区中全局地址[15:0]以内的任意值，参照闪存扇区大小的概览	

当清空 FSTAT[CCIF]来执行擦除 Flash 扇区指令编写时，存储控制器将会擦除被选中的扇区并且会校验已被擦除的部分。当擦除 Flash 扇区指令完成后 FSTAT[CCIF]标志位将会被设置。当擦除 Flash 扇区时，错误标志位 FSTAT[ACCERR]置 1，错误的状态如下：①执行指令时 CCOBIX[2:0]不等于 001；②当前模式下不能获取指令；③提供了一个无效的地址；④提供了不一致的长字地址（全局地址[1:0]!=00）。



9.3　Flash 驱动构件封装要点分析及实现

9.3.1　Flash 驱动构件封装要点

Flash 使用过程首先应予以模块的初始化，其次应具有擦除指定扇区、向指定扇区及地址写入特定长度的数据两种基本操作。按照构件的思想，可将它们封装成三个独立的功能函数，初始化函数完成对 Flash 模块的工作属性的设定，Flash 擦除、写入函数则完成实际的擦除、写入任务。

1．初始化函数 void flash_init（void）

在操作 Flash 模块前，需要对模块进行初始化，主要是包括配置时钟、判断并等待 Flash 操作命令完成、清相关标志位、杂项模块中平台控制寄存器的 PLACR_ESFC 的设置。

2．擦除函数 uint_8 flash_erase_sector（uint_16 sectorNo）

由于在写入之前 Flash 字节或者长字节必须处于擦除状态（不允许累积写入），因此在写入操作前，一般先进行 Flash 的擦除操作。

擦除操作有擦除块（擦除 Flash 中所有的地址）和擦除扇区两种操作模式，在本书中，主要以擦除扇区为例。首先需要确定要擦除的扇区号，作为参数传入，最终返回擦除状态（正常/异常）。函数过程中主要利用索引寄存器和参数寄存器来控制擦除的命令、地址范围及擦除数据（写 0xFF）

3．写入函数 uint_8 flash_write（uint_16 sectNo,uint_16 offset,uint_16 cnt,uint_8 buf[]）

写入函数与擦除函数类似，主要区别在于，擦除操作向目标地址中写 0xFF，而写入操作需要写入指定数据，因此写入操作的入口参数较多，包括目标扇区号、写入扇区内部偏移地址、写入字节数目，以及源数据缓冲区首地址，写入后返回写入状态（正常/异常）。

9.3.2 Flash 驱动构件的实现

1．Flash 驱动构件.c 文件

```
//包含头文件
#include "flash.h"
#include "uart.h"

//=================内部调用函数声明=========================================
//==========================================================================
//函数名称：flash_cmd_launch
//函数返回：0=成功；1=失败
//参数说明：无
//功能概要：启动 Flash 命令
//==========================================================================
static uint_32 flash_cmd_launch(void);
//==========================================================================

//=================外部接口函数=============================================
//==========================================================================
//函数名称：flash_init
//函数返回：无
//参数说明：无
//功能概要：初始化 Flash 模块
//==========================================================================
void flash_init(void)
{
    //配置时钟分频寄存器
    FTMRE_FCLKDIV |=FTMRE_FCLKDIV_FDIV(0x17);   //总线时钟 24 MHz

    //等待命令完成
```

```
    while(!(FTMRE_FSTAT & CCIF));

    //清除访问出错标志位
    FTMRE_FSTAT = ACCERR | FPVIOL;

    //置杂项模块中平台控制寄存器的 PLACR_ESFC，Flash 模块 擦写保护
    BSET(MCM_PLACR_ESFC_SHIFT,MCM_PLACR);

    //清杂项模块中平台控制寄存器的 PLACR_ESFC，Flash 模块擦写不保护
    BCLR(MCM_PLACR_ESFC_SHIFT,MCM_PLACR);   //（实验观察现象用）

}
//=============================================================================
//函数名称：flash_erase_sector
//函数返回：函数执行执行状态：0=正常；非 0=异常
//参数说明：sectorNo：扇区号（KEA128 实际使用 0～255）每个扇区 512 B
//功能概要：擦除指定 Flash 扇区
//=============================================================================
uint_8 flash_erase_sector(uint_16 sectorNo)
{
    union
    {
        uint_32  word;
        uint_8   byte[4];
    }dest;

    dest.word    = (uint_32)(sectorNo*(1<<9)); //512 B

    while(!(FTMRE_FSTAT & CCIF));

    if(FTMRE_FSTAT & (ACCERR | FPVIOL))
        FTMRE_FSTAT |= ACCERR | FPVIOL;
    //若擦除 sector0 时，则解锁设备
    if(dest.word < 0x200)
    {
        //写入 4 B
        FTMRE_FCCOBIX = FTMRE_FCCOBIX_CCOBIX(0b000);
        FTMRE_FCCOBHI = 0x06;                    //PGM4=0x06
        //设置目标地址
        FTMRE_FCCOBLO = 0x00;

        FTMRE_FCCOBIX = FTMRE_FCCOBIX_CCOBIX(0b001);
        FTMRE_FCCOBHI = 0x04;
        FTMRE_FCCOBLO = 0x0C;

        //数据
        FTMRE_FCCOBIX = FTMRE_FCCOBIX_CCOBIX(0b010);
```

```
        FTMRE_FCCOBHI = 0xFF;
        FTMRE_FCCOBLO = 0xFF;

        FTMRE_FCCOBIX = FTMRE_FCCOBIX_CCOBIX(0b011);
        FTMRE_FCCOBHI = 0xFF;
        FTMRE_FCCOBLO = 0xFF;
        //执行命令序列
        if(1 == flash_cmd_launch())     //若执行命令出现错误
            return 2;                   //解锁命令错误
    }
    else
    {
        //设置擦除命令
        FTMRE_FCCOBIX = FTMRE_FCCOBIX_CCOBIX(0b000);
        FTMRE_FCCOBHI = ERSSCR;         //擦除扇区命令(通用命令寄存器（高）)

        //设置目标地址
        FTMRE_FCCOBLO = dest.byte[2];

        FTMRE_FCCOBIX = FTMRE_FCCOBIX_CCOBIX(0b001);
        FTMRE_FCCOBHI = dest.byte[1];
        FTMRE_FCCOBLO = dest.byte[0];

        //执行命令序列
        if(1 == flash_cmd_launch())     //若执行命令出现错误
            return 1;                   //擦除命令错误
    }

    return 0;                           //成功返回
}

//=========================================================================
//函数名称：flash_write
//函数返回：函数执行状态，0=正常；非 0=异常
//参数说明：sectNo：目标扇区号（KEA128 实际使用 0～255）每个扇区 512 B
//         offset:写入扇区内部偏移地址（0～508）
//         cnt：写入字节数目（4～512）
//         buf：源数据缓冲区首地址
//功能概要：Flash 写入操作
//=========================================================================
uint_8 flash_write(uint_16 sectNo,uint_16 offset,uint_16 cnt,uint_8 buf[])
{
    uint_32 size;

    uint_32 destaddr;
    union
    {
```

```
        uint_32   word;
        uint_8  byte[4];
    } dest;

    if(offset%4 != 0)
        return 1;   //参数设定错误，偏移量未对齐（4 字节对齐）

    while(!(FTMRE_FSTAT & CCIF));

    if(FTMRE_FSTAT & (ACCERR | FPVIOL))
        FTMRE_FSTAT |= ACCERR | FPVIOL;

    //设置写入命令
    FTMRE_FCCOBIX = FTMRE_FCCOBIX_CCOBIX(0b000);
    FTMRE_FCCOBHI = 0x06;

    destaddr = (uint_32)(sectNo*(1<<9) + offset);      //计算地址
    dest.word = destaddr;
    for(size=0; size<cnt; size+=4, dest.word+=4, buf+=4)
    {
        //设置目标地址
        FTMRE_FCCOBLO = dest.byte[2];

        FTMRE_FCCOBIX = FTMRE_FCCOBIX_CCOBIX(0b001);
        FTMRE_FCCOBHI  = dest.byte[1];
        FTMRE_FCCOBLO  = dest.byte[0];

        //复制数据
        FTMRE_FCCOBIX = FTMRE_FCCOBIX_CCOBIX(0b010);
        FTMRE_FCCOBHI  = buf[1];
        FTMRE_FCCOBLO  = buf[0];

        FTMRE_FCCOBIX = FTMRE_FCCOBIX_CCOBIX(0b011);
        FTMRE_FCCOBHI  = buf[3];
        FTMRE_FCCOBLO  = buf[2];

        if(1 == flash_cmd_launch())
            return 2;                                  //写入命令错误
    }

    return 0;                                          //成功执行
}
//=========================================================================
//函数名称：flash_secure
//函数返回：函数执行状态；0=正常；非 0=异常
//参数说明：无
//功能概要：Flash 加密操作
```

```
//=========================================================================
uint_8 flash_secure(uint_8 key[])
{
    uint_8 vecdata[128];
    uint_16 i;
    uint_8* dataaddr;
    dataaddr=(uint_8*)0x400;
    for(i=0;i<8;i++)
        vecdata[i]=key[i];
        vecdata[8]=0xFF;            //不设置块保护
        vecdata[9]=0xFF;            //不设置块保护
        vecdata[10]=0xFF;           //不设置块保护
        vecdata[11]=0xFF;           //不设置块保护
        vecdata[12]=0xFF;           //Flash 选项寄存器默认设置为 0xff
        vecdata[13]=0x80;           //加密芯片，设置允许后门秘钥访问 0x0D0
        vecdata[14]=0xFF;           //保护寄存器内容
        vecdata[15]=0xFF;           //保留字段
     for(i=16;i<128;i++)
        vecdata[i]=dataaddr[i];     //复制原来 Flash 中的内容
    union
    {
        uint_32   word;
        uint_8  byte[4];
    } dest;

    //设置写入命令
    FTMRE_FCCOBIX = FTMRE_FCCOBIX_CCOBIX(0b000);
    FTMRE_FCCOBHI = 0x06;           //PGM4=0x06
    dest.word = 0x400;
    dataaddr=vecdata;
    for(i=0; i<128; i+=4, dest.word+=4,dataaddr=dataaddr+4)
    {
        //设置目标地址
        FTMRE_FCCOBLO = dest.byte[2];

        FTMRE_FCCOBIX = FTMRE_FCCOBIX_CCOBIX(0b001);
        FTMRE_FCCOBHI  = dest.byte[1];
        FTMRE_FCCOBLO  = dest.byte[0];

        //复制数据
        FTMRE_FCCOBIX = FTMRE_FCCOBIX_CCOBIX(0b010);
        FTMRE_FCCOBHI  = dataaddr[1];
        FTMRE_FCCOBLO  = dataaddr[0];

        FTMRE_FCCOBIX = FTMRE_FCCOBIX_CCOBIX(0b011);
        FTMRE_FCCOBHI  = dataaddr[3];
        FTMRE_FCCOBLO  = dataaddr[2];
```

```
        if(1 == flash_cmd_launch())
            return 2;  //写入命令错误
    }
        return 0;
}
//=========================================================================
//函数名称：flash_unsecure
//函数返回：函数执行状态，0=正常；非 0=异常
//参数说明：无
//功能概要：删除秘钥，解密芯片
//=========================================================================
uint_8 flash_unsecure()
{
    uint_8 vecdata[128];
    uint_16 i;
    uint_8* dataaddr;
    dataaddr=(uint_8*)0x400;
    for(i=0;i<8;i++)
        vecdata[9]=0xFF;              //不设置块保护
        vecdata[10]=0xFF;             //不设置块保护
        vecdata[11]=0xFF;             //不设置块保护
        vecdata[12]=0xFF;             //Flash 选项寄存器默认设置为 0xff
        vecdata[13]=0xFE;             //解密芯片 0x0D0
        vecdata[14]=0xFF;             //Flash 保护寄存器
        vecdata[15]=0xFF;             //保留字段
    for(i=16;i<128;i++)
        vecdata[i]=dataaddr[i];       //复制原来 Flash 中的内容
    union
    {
        uint_32   word;
        uint_8    byte[4];
    }dest;
    //设置写入命令
    FTMRE_FCCOBIX = FTMRE_FCCOBIX_CCOBIX(0b000);
    FTMRE_FCCOBHI = 0x06;             //PGM4=0x06
    dest.word = 0x400;
    dataaddr=vecdata;
    for(i=0; i<128; i+=4, dest.word+=4,dataaddr=dataaddr+4)
    {
        //设置目标地址
        FTMRE_FCCOBLO = dest.byte[2];

        FTMRE_FCCOBIX = FTMRE_FCCOBIX_CCOBIX(0b001);
        FTMRE_FCCOBHI  = dest.byte[1];
        FTMRE_FCCOBLO  = dest.byte[0];
```

```
        //拷贝数据
        FTMRE_FCCOBIX = FTMRE_FCCOBIX_CCOBIX(0b010);
        FTMRE_FCCOBHI  = dataaddr[3];
        FTMRE_FCCOBLO  = dataaddr[2];

        FTMRE_FCCOBIX = FTMRE_FCCOBIX_CCOBIX(0b011);
        FTMRE_FCCOBHI  = dataaddr[1];
        FTMRE_FCCOBLO  = dataaddr[0];

        if(1 == flash_cmd_launch())
            return 2;  //写入命令错误
    }
    return 0;
}

//=========================================================================
//函数名称：flash_protect
//函数返回：函数执行状态，0=正常；非 0=异常。
//参数说明：size：保护扇区数（从 0x0000 地址开始）
//功能概要：Flash 保护操作
//说    明：进行扇区保护时，扇区保护的范围只能增加，不能减少
//          目前程序中保护扇区块数只能为 0、4、8、16、32、256，其中 0 为不保护；4、8、16、
//          32 为低地址域保护（相互之间不能切换，只能执行一次）；256 为全保护
//=========================================================================
void flash_protect(uint_16 size)
{
    switch(size)
    {
    case 0:     //不保护
        //FTMRE_FPROT 寄存器中 FPOPEN、FPHDIS、FPLDIS 分别为第 7、5、2 位，FPLS 为第
          1 和 0 位
        FTMRE_FPROT = (1<<7) | (1<<5) | (1<<2);
        break;
    case 4:     //保护开始的 2 KB(0x00000～0x007FF)  FPLS=0b00 时,保护区域 2 KB
        FTMRE_FPROT = (1<<7) | (1<<5) | (0<<2)|(0<<1) | (0<<0);
        break;
    case 8:     //保护开始的 4 KB(0x00000～0x00FFF)   FPLS=0b01 时,保护区域 4 KB
        FTMRE_FPROT = (1<<7) | (1<<5)| (0<<2)|(0<<1) | (1<<0);
        break;
    case 16:     //保护开始的 8 KB(0x00000～0x01FFF)  FPLS=0b10 时,保护区域 8 KB
        FTMRE_FPROT = (1<<7) | (1<<5)| (0<<2)|(1<<1) | (0<<0);
        break;
    case 32:     //保护开始的 16 KB(0x00000～0x03FFF)  FPLS=0b11 时,保护区域 16 KB
        FTMRE_FPROT = (1<<7) | (1<<5)| (0<<2)|(1<<1) | (1<<0);   //有问题
        break;
//  case 224:  //保护除开始 16 KB 之外的 112 KB(0x04000～0x1FFFF)  FPLS=0b11 时,不保
                 护区域 16 KB
```

```
//      FTMRE_FPROT = (0<<7) | (1<<5)| (0<<2)|(1<<1) | (1<<0);
//      break;
    case 256:     //全保护
        FTMRE_FPROT = (0<<7) | (1<<5)| (1<<2);
        break;
    default:
        break;
    }
}

//=========================================================================
//函数名称：flash_cmd_launch
//函数返回：0=成功；1=失败
//参数说明：无
//功能概要：启动 Flash 命令
//=========================================================================
static uint_32 flash_cmd_launch(void)
{
    int i=200000;

    //启动命令
    FTMRE_FSTAT |= CCIF;

    while(i--);      //不加延时会复位

    //等待命令结束
    while(!(FTMRE_FSTAT & CCIF));

    //检查错误标志
    if(FTMRE_FSTAT & (ACCERR | FPVIOL | MGSTAT0))
        return 1 ;   //执行命令出错
    return 0;        //执行命令成功
}
//=========================================================================
```

2．Flash 驱动构件测试工程主函数

```
//说明见工程文件夹下的 Doc 文件夹内 Readme.txt 文件
#define GLOBLE_VAR  //只需在 main.c 中定义一次，用来防止全局变量的重复定义

#include "includes.h"

int main(void)
 {
    //1. 声明主函数使用的变量
    uint8_t revstat,cmd,state;
    uint16_t i,j,num,tem;
```

```
    uint16_t count,count2;
    uint32_t page;
    uint8_t  rev[512],writebuf[512];
    uint32_t len,pstartadd;
    //2. 关总中断
    DISABLE_INTERRUPTS;                 //关总中断
    //3. 初始化外设模块
    uart_init (UART_Debug,9600);        //初始化"调试串口"
    uart_init (UART_0,9600);
    uart_init (UART_1,9600);
    uart_init (UART_2,9600);
    light_init(LIGHT_0, LIGHT_ON);      //初始化灯 0
    printf("Hello Flash!\r\n");         //串口发送初始化提示
    printf("Press ? to get help\r\n");
    flash_init();                       //Flash 初始化
    flash_protect(SIZE_16K);            //防止程序区被擦除，直接将前 32 个扇区保护
    //4. main 中使用的局部变量初始化
    revstat   = 1;
    tem = 0;
    //5. 全局变量初始化
    //6. 使能模块中断
    //7. 开总中断
    ENABLE_INTERRUPTS;                  //开总中断

    //进入主循环
    //主循环开始=============================================================
    for (;;)
    {
        len = 0;
        count = 0;
        count2= 0;
        page = 0;
        num = 0;
        pstartadd = 0;
        i = 0;
        while(revstat)
            cmd = uart_re1(UART_2,&revstat);
        revstat = 1;

        switch (cmd)
        {
case '?'://命令帮助
            printf("擦除,读取,写入格式如下:\r\n");
            printf("   擦除 E:扇区数(如 E:50)\r\n");
            printf("   写入 W:扇区数:开始字节:写入字节数:写入的数据(如 W:50:0:
                   4:1,2,3,4)\r\n");
            printf("   读取 R:扇区数:开始字节:读取字节数 (如 R:50:0:4)\r\n");
```

```
        printf("   保护 P:大小 (如全保护 P:256)，为了防止程序部分被擦除，前 32
               扇区已保护，此处只可全保护\r\n");
        printf("          详细介绍见 Readme.txt\r\n");
        printf("注意:可用扇区(0～255),扇区地址空间(0～511)\r\n");
        break;
    //擦除本程序中"中断向量表+Flash 配置域+代码区"的大小约为 10K 多，建议靠后擦除，
    case 'E':
    case 'e':
        //跳过冒号
        cmd = uart_re1(UART_2,&revstat);
        if(cmd==':')
        {
            //接收串口数据
            while(revstat == 0)
            {
                rev[count++] = uart_re1(UART_2,&revstat);
            }
            //计算地址
            while(i<(count-1))
            {
                page = page*10 +rev[i++]-'0';
            }
            //擦除 Flash
            state = flash_erase_sector(page);
            if(state == 0)
                printf(" Erase succeed\r\n");
            else
                printf(" Erase failed\r\n");
        }
        break;

    case 'R': //读取
    case 'r':
        //跳过冒号
        cmd = uart_re1(UART_2,&revstat);
        i = 0;
        //接收串口数据
        while(revstat == 0)
        {
            rev[count++] = uart_re1(UART_2,&revstat);
        }
        //计算页号
        while(rev[i] != ':')
        {
            page = page*10 +rev[i++]-'0';
        }
        i++;
```

```
                //计算开始地址
                while(rev[i] != ':')
                {
                    pstartadd = pstartadd*10 +rev[i++]-'0';
                }
                i++;
                //计算读的字节个数
                while((rev[i] >= '0')&&(rev[i] <= '9')&&(i<count))
                {
                    len = len*10+rev[i++]-'0';
                }
                printf("R:");
                //printf("%s",(uint8_t*)(page*512+pstartadd));
                printf("len=%d\n",len);
                printf("addr=%d\n",page*512+pstartadd);

                uart_sendN(UART_2, len, (uint8_t*)(page*512+pstartadd));
                printf("\r\n");
                break;
            case 'W'://写入
            case 'w':
                //跳过冒号
                cmd = uart_re1(UART_2,&revstat);
                i = 0;
                //接收串口数据
                while(revstat == 0)
                {
                    rev[count++] = uart_re1(UART_2,&revstat);
                }
                //计算页号
                while(rev[i] != ':')
                {
                    page = page*10+rev[i++]-'0';
                }
                i++;
                //计算页内开始写的偏移地址
                while(rev[i] != ':')
                {
                    pstartadd = pstartadd*10+rev[i++]-'0';
                }
                i++;
                //计算要写的字节个数
                while(rev[i] != ':')
                {
                    len = len*10+rev[i++]-'0';
                }
                i++;
                //取得要写入的数据
```

```
            j = 0;
            while(i<(count-1))
            {
                if(rev[i] == ',')
                {
                    i++;
                }
                else
                {
                    writebuf[j] = rev[i];
                    j++; i++;
                }
            }
            //写入 Flash
            state = flash_write(page,pstartadd,len,writebuf);
            if(0 == state)
                printf(" Write succeed\r\n");
            else
                printf(" The areas are protected\r\n");
            break;
        case 'P'://保护
        case 'p':
            //跳过冒号
            cmd = uart_re1(UART_2,&revstat);
            i = 0;
            //接收串口数据
            while(revstat == 0)
            {
                rev[count2++] = uart_re1(UART_2,&revstat);
            }
            //计算保护区域的大小
            while(i<(count2-1))
            {
                num = num*10 +rev[i++]-'0';
            }

            if(0 == tem)  //判断是否全部保护
            {
                if( 256 == num )
                {
                    //保护 Flash
                    flash_protect(num);
                    printf(" Protect succeed\r\n");
                    tem =1;
                }
                else
                    printf(" All areas need to be protected\r\n");
            }
```

```
                else
                   printf(" All areas have been protected\r\n");
                break;

            default:
                break;
           }//end_switch

    }  //end_for

    //主循环结束=================================================================
    return 0;
}
```

3．Flash 测试工程

测试工程功能概述如下。

（1）串口通信格式：波特率 9600，1 位停止位，无校验。

（2）上电或按复位按钮时，调试串口输出“Hello Flash!\nPress ? to get help”，按“?”键可获得提示帮助。

（3）主循环中，LIGHT_0 小灯状态不变，长亮。

（4）串口作为命令行输入，用于解析 flash 操作命令，实现对 Flash 扇区的擦除、写、读及保护操作。

Falsh 构件的测试工程位于网上光盘中的“..\program\CH09_KEA128-flash”文件夹。运行样例程序测试，PC 机串口与 KEA128 芯片进行交互步骤如下。

（1）正确下载程序，复位芯片后，可通过 PC 上的串口调试工具看到以下提示信息。

```
================================================================================
Hello Flash!
================================================================================
```

（2）从 PC 通过串口向 KEA128 发送字符“?（英文格式）”，出现提示菜单：

```
================================================================================
擦除,读取,写入格式如下:
擦除 E:扇区数(如 E:14)，本程序建议从 14 扇区开始擦除，否则会擦除程序区;
读取 R:扇区数:开始字节:读取字节数 (如 R:14:0:4)\r\n");
写入 W:扇区数:开始字节:写入字节数:写入的数据(如 W:14:0:4:1,2,3,4)\r\n");
保护 P:大小 (如全保护 P:256)，保护块数只能为：0、4、8、16、32、224、256;
4、8、16、32 为低地址域保护（相互之间不能切换）；224 为保护后 112K;
注意:可用扇区(0～255),扇区地址空(0～511)\r\n");
================================================================================
```

（3）根据提示，依次进行对 Flash 的擦除、写入、读取等操作。

测试如下命令和操作参数：E:14；W:14:0:4:1,2,3,4；R:14:0:4

运行样例程序如图 9-2 所示。

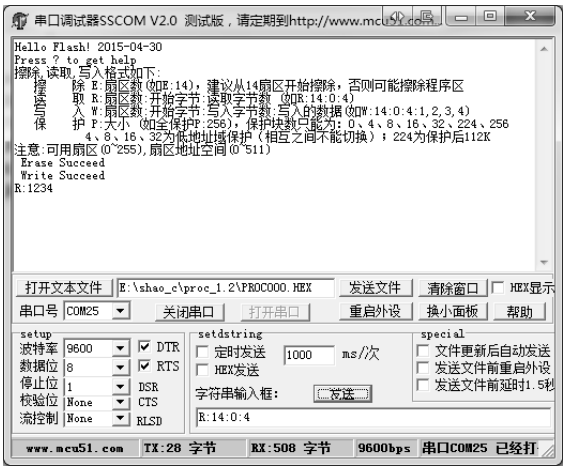


图 9-2　运行 Flash 在线编程样例程序

9.4　Flash 模块的保护与加密

9.4.1　Flash 模块的保护

1．Flash 存储保护介绍

可以通过设置 FPROT 寄存器来保护 Flash 不受意外写入或擦除的影响，Flash 有三个独立的存储域，其中一个是从 Flash 全局地址 0x0000 处向上的区域，称为低址域；另一个是从 Flash 全局地址 0x7FFF 处向下的区域，称为高址域；最后是闪存中剩余的地址区域，可以激活用来保护 Flash。

Flash 保护字节内容在 FPROT 寄存器中，保护功能取决于 FPROT 寄存器中位的配置。

FPROT 寄存器中 FPOPEN、FPHDIS、FPLDIS 位的含义在此不加赘述，请参见 9.2.1 节相关内容。表 9-4 简洁明了地阐述了对 Flash 存储模块保护时的 FPROT 寄存器的设置。

表 9-4　Flash 保护功能

FPOPEN	FPHDIS	FPLDIS	功　能①
1	1	1	无 Flash 保护
1	1	0	保护低地址范围
1	0	1	保护高地址范围
1	0	0	保护高低地址范围
0	1	1	闪存满保护
0	1	0	未受保护的低地址范围
0	0	1	未受保护的高地址范围
0	0	0	未受保护的高低地址范围

例如，当 FPOPEN=0 时，整个 Flash 是处于全保护的初始状态（包括高址域、低址域及

① 范围请参照 KEA128 参考手册中表 18-9 和表 18-10

剩余其他地址区域），而通过 FPHDIS 和 FPLDIS 定义从全保护范围中剔除掉（即不保护）的地址域（高址域/低址域），而值为 0（启用）的位对应的区域则不受保护，并通过 FPHS 和 FPLS 位设置对应范围。

当 FPOPEN=1 时，整个 Flash 处于全不保护的初始状态（包括高址域、低址域及剩余其他地址区域），通过 FPHDIS 和 FPLDIS 定义从全不保护范围中剔除掉（即保护）的地址域（高址域/低址域），而值为 0（启用）的位对应的区域则受保护，并通过 FPHS 和 FPLS 位设置对应范围。

在单芯片模式中，需要重新写入或提供尽可能多保护的应用程序可以使用 Flash 保护方案。在有些保护方案之间是可以进行转换的，基本原则是保护区域只能增加而不能减少。关于保护方案及转换，在本书中将不加详细介绍，如有需要请参见《KEA128 系列用户参考手册》18.3.6 保护。表 9-5 和表 9-6 列出了 Flash 保护的地址范围。

表 9-5 Flash 保护的高址范围

FPHS[1:0]	全局地址范围	保护的大小
00	0x7C00～0x7FFF	1 KB
01	0x7800～0x7FFF	2 KB
10	0x7000～0x7FFF	4 KB
11	0x6000～0x7FFF	8 KB

表 9-6 Flash 保护的低址范围

FPLS[1:0]	全局地址范围	保护的大小
00	0x0000～0x07FF	2 KB
01	0x0000～0x00FF	4 KB
10	0x0000～0x01FF	8 KB
11	0x0000～0x03FF	16 KB

2．Flash 存储保护驱动构件的实现

根据有关 Flash 存储保护介绍，设计 Flash 存储保护的驱动构件。首先在头文件中，加入以下对于待保护扇区大小的宏定义。一般情况下，Flash 保护的主要任务有两种，一是保护程序区，处于 Flash 存储模块 0x0000 开始的低址域范围（一般 16 KB 以内）；二是对于经常更新代码的 Flash，保护除程序区以外的数据区。因此在设计本构件时，并未将所有保护情况完全罗列，主要保护低地址域的 0、4、8、16、32 个扇区、除了前 16 KB 之外的 112 KB 区域，以及保护所有扇区。

```
#define NONE_SIZE        0
#define SIZE_2K          4
#define SIZE_4K          8
#define SIZE_8K          16
#define SIZE_16K         32
#define LAST_112K        224    //除了前 16 KB 之外的 112 KB
#define ALL_SIZE         256

//=============================================================================
//函数名称：flash_protect
//函数返回：函数执行状态，0=正常；非 0=异常。
//参数说明：size：保护扇区数（从 0x0000 地址开始）
//功能概要：Flash 保护操作
//=============================================================================
void flash_protect(uint_16 size);
```

向 Flash 驱动构件 flash.c 文件中添加 Flash 存储保护驱动相关代码，构件如下：

```
//=====================================================================
//函数名称：flash_protect
//函数返回：函数执行状态，0=正常；非 0=异常。
//参数说明：size：保护扇区数（从 0x0000 地址开始）
//功能概要：Flash 保护操作
//说    明：进行扇区保护时，扇区保护的范围只能增加，不能减少
//          目前程序中保护扇区块数只能为：0、4、8、16、32、256，其中 0 为不保护、4、8、16、32 为低
//          地址域保护（相互之间不能切换，只能执行一次），256 为全保护。
//=====================================================================
void flash_protect(uint_16 size)
{
    switch(size)
    {
    case 0:     //不保护
        //FTMRE_FPROT 寄存器中 FPOPEN、FPHDIS、FPLDIS 分别为第 7、5、2 位，FPLS 为第
          1 和 0 位
        FTMRE_FPROT = (1<<7) | (1<<5) | (1<<2);
        break;
    case 4:     //保护开始的 2 KB(0x00000～0x007FF)  FPLS=0b00 时,保护区域 2 KB
        FTMRE_FPROT = (1<<7) | (1<<5) | (0<<2)|(0<<1) | (0<<0);
        break;
    case 8:     //保护开始的 4 KB(0x00000～0x00FFF)   FPLS=0b01 时,保护区域 4 KB
        FTMRE_FPROT = (1<<7) | (1<<5)| (0<<2)|(0<<1) | (1<<0);
        break;
    case 16:     //保护开始的 8 KB(0x00000～0x01FFF)  FPLS=0b10 时,保护区域 8 KB
        FTMRE_FPROT = (1<<7) | (1<<5)| (0<<2)|(1<<1) | (0<<0);
        break;
    case 32:     //保护开始的 16 KB(0x00000～0x03FFF)  FPLS=0b11 时,保护区域 16 KB
        FTMRE_FPROT = (1<<7) | (1<<5)| (0<<2)|(1<<1) | (1<<0);   //有问题
        break;
//case 224:  //保护除开始 16 KB 之外的 112 KB(0x04000～0x1FFFF)  FPLS=0b11 时,不
               保护区域 16 KB
//      FTMRE_FPROT = (0<<7) | (1<<5)| (0<<2)|(1<<1) | (1<<0);
//   break;
    case 256:     //全保护
        FTMRE_FPROT = (0<<7) | (1<<5)| (1<<2);
        break;
    default:
        break;
    }
}
```

在 Flash 驱动构件测试工程中可以进行 Flash 扇区保护的测试，如在串口调试工具窗口中输入“P:256”进行 Flash 全地址范围保护后，再进行任何擦除操作都是不允许的，窗口将返回提示“The Areas are protected!”。

9.4.2 Flash 模块的安全

Flash 的安全状态是由 FSEC[SEC]定义的，在复位期间，Flash 模块从 Flash 配置区读取安全字节来初始化 FSEC 寄存器。如果 MCU 从一个 Flash 擦除和编程指令可用的模式开始，并且 Flash 的顶端域不被保护，可以通过对安全字节执行写操作来改变复位之后的安全状态。一旦 Flash 安全字节写操作完成后，它的新值将会在下一次 MCU 复位后起作用。

1．芯片加密

软件操作不当、硬件连接错误都有可能导致芯片加密，具体请详见网上光盘中的《KEA128 快速应用指南》。

2．芯片解密

1）使用后门密钥解密

可以使用后门密钥使 MCU 处于非安全状态，这项技术需要后门密钥内容的相关知识，在本书中不加过多描述，详细请参见《KEA128 系列用户参考手册》18.3.6 节。

2）使用 SWD 解密

通过下述的方法擦除 Flash 可以使 MCU 处于不安全状态。

（1）使用 RESET 引脚或 DAP_CTRL[3]来复位 MCU。

（2）通过 SWD 设置 DAP_CTRL[0]位调用执行批量擦除。

（3）通过 SWD 解除 RESET 引脚或 DAP_CTRL[3]来释放复位。

（4）等待直到 DAP_CTRL[0]位清 0（在批量擦除完成后，DAP_CTRL[0]位自动清 0）。这时，CPU 处于阻塞状态，并且 MCU 处于非安全状态（Flash 配置域中的 Flash 安全字节被写入为 0xFE）。

（5）复位 MCU。

3）使用 JLink 软件进行解密

在实际应用过程中，使用 JLink 软件进行解密更加快速方便。具体操作步骤请详见网上光盘中的《KEA128 快速应用指南》。

第 10 章

ADC 与 ACMP 模块

本章导读：本章阐述 KEA128 的 A/D 转换模块（ADC）和比较器模块（ACMP）构件化编程。主要内容有：10.1 节和 10.2 节介绍 A/D 转换模块，10.1 节在简要阐述 ADC 编程要素基础上，给出 KEA128 芯片 ADC 知识要素及技术要点，给出 ADC 构件接口函数说明及使用方法举例，这样就可以对 KEA128 芯片 ADC 模块进行编程操作；10.2 节给出 ADC 驱动构件的制作方法；10.3 节和 10.4 节介绍比较器模块，10.3 节在简要阐述 ACMP 编程要素基础上，给出 KEA128 芯片 ACMP 知识要素及技术要点，给出 ACMP 构件接口函数说明及使用方法举例，这样就可以对 KEA128 芯片 ACMP 模块进行编程操作了；10.4 节给出 ACMP 驱动构件的制作方法。期望通过这一章的学习，掌握嵌入式系统 ADC 和 ACMP 程序的设计。

本章参考资料：本章 10.3 节来源 KEA128 参考手册的第 24 章 AD 模块寄存器，第 3 章芯片资源配置参考自《KEA128 数据手册》的 5.4.1 节；本章 10.5 节参考自《KEA128 参考手册》的第 25 章（ACMP 模块寄存器）。

10.1 KEA128 芯片 ADC 驱动构件及使用方法

10.1.1 ADC 编程知识要素

A/D 转换模块（Analog To Digital Convert Module）即模/数转换模块，其功能是将电压信号转换为相应的数字信号。实际应用中，这个电压信号可能由温度、湿度、压力等实际物理量经过传感器和相应的变换电路转化而来。经过 A/D 转换后，MCU 就可以处理这些物理量。进行 A/D 转换，应该了解一些基本问题：第一，采样精度是多少；第二，采样速率有多快；第三，滤波问题；第四，物理量回归等。

1．采样精度

采样精度就是指数字量变化一个最小量时模拟信号的变化量，即我们通常所说的采样位数。通常，MCU 的采样位数为 8、10 位，某些增强型的可达到 12 位，而专用的 A/D 采样芯片则可达到 12 位、14 位，甚至 16 位。设采样位数为 N，则最小的能检测到的模拟量变化值为 $1/2^N$。

2．采样速率

采样速率是指完成一次 A/D 采样所要花费的时间，在多数的 MCU 中要花费大于 15～20 个指令周期，因而此速率和所选器件的工作频率有很大的关系。

3．滤波

为了使采样的数据更准确，必须对采样的数据进行筛选去掉误差较大的毛刺。通常我们采用中值滤波和均值滤波来提高采样精度。所谓中值滤波，就是将 *M* 次连续采样值按大小进行排序，取中间值作为滤波输出。而均值滤波，是把 *N* 次采样结果值相加，然后再除以采样次数 *N*，得到的平均值就是滤波结果。若要得到更高的精度，可以通过建立其他误差模型分析方式来实现。

4．物理量回归

在实际应用中，得到稳定的 A/D 采样值以后，还需要把 A/D 采样值与实际物理量对应起来，这一步称为物理量回归。A/D 转换的目的是把模拟信号转化为数字信号，供计算机进行处理，但必须知道 A/D 转换后的数值所代表的实际物理量的值，这样才有实际意义。例如，利用 MCU 采集室内温度，A/D 转换后的数值是 126，实际它代表多少温度呢？如果当前室内温度是 25.1℃，则 A/D 值 126 就代表实际温度 25.1℃。

10.1.2 KEA128 芯片的 ADC 引脚与通道号

KEA128 芯片的 A/D 转换模块转换电压范围为 2.7～5.5 V；它使用了线性逐次逼近算法，共有 16 个引脚可以作为 A/D 转换的外部输入引脚；支持的具有 8、10、12 位共 3 种精度的转换模式；可以进行单次或连续转换（在单次转换后自动返回空闲状态）；支持高达 8 个可选大小的 FIFO 结果队列；还支持多达 4 个输入时钟源。KEA128 的 A/D 转换模块不具有差分输入引脚功能。

KEA128 芯片的 A/D 转换模块的单次转换消耗的时间由选择的时钟源和分频率决定。例如，在 10 位分辨率模式中，总线时钟选择作为输入时钟源，输入时钟 1 分频，总线频率为 8 MHz，根据参考手册，单次转换的转换时间约为 3.5 μs。

KEA128 芯片具有一个 ADC 模块，记为 ADC0。由于 ADC 模块的 ADC 通道的数量由专门的芯片封装引出脚决定，本书选用 LQFP 封装 80 引脚的 SKEA128 芯片，具有多达 16 个外部模拟输入 5 个内部模拟输入，包括内部带隙参考电路、温度传感器和参考电压；另外，还有其他形式的模拟输入通道，如 ADC 模块内包含一个温度传感器，它的输出信号接在 ADC 模拟量输入通道上，通道号位 22。ADC 输入通道情况如表 10-1 所示。

表 10-1 ADC 通道分配

ADCH	通　道	输　入
00000	AD0	PTA0/ADP0
00001	AD1	PTA1/ADP1
00010	AD2	PTA6/ADP2
00011	AD3	PTA7/ADP3
00100	AD4	PTB0/ADP4
00101	AD5	PTB1/ADP5
00110	AD6	PTB2/ADP6
00111	AD7	PTB3/ADP7
01000	AD8	PTC0/ADP8

续表

ADCH	通　　道	输　　入
01001	AD9	PTC1/ADP9
01010	AD10	PTC2/ADP10
01011	AD11	PTC3/ADP11
01100	AD12	PTF4/ADP12
01101	AD13	PTF5/ADP13
01110	AD14	PTF6/ADP14
01111	AD15	PTF7/ADP15
10000～10011	AD16～AD19	Vss
10100～10101	AD20～AD21	保留
10110	AD22	温度传感器
10111	AD23	带隙
11000～11100	AD24～AD28	保留
11101	AD29	VREFH
11110	AD30	VREFL
11111	模块禁用	无

10.1.3　KEA128 芯片 ADC 构件头文件及使用方法

头文件 adc.h 中主要介绍了 4 个函数原型的声明。这 4 个函数给出了 ADC 驱动构件对上层构件或软件所提供的接口函数，分别为初始化函数 adc_init 的通道号、采样精度被设计为函数参数，读取通道转换值 adc_readonce 函数、参数为通道号，软件均值滤波 adc_mid 函数对采集的 A/D 值进行均值滤波，以及软件中值滤波 adc_ave 函数对采集的 A/D 值进行中值滤波。这样的话，应用程序和上层构件在使用（调用）它时，将具有极大的灵活性。

1．ADC 驱动构件头文件（adc.h）

```
//=============================================================================
//文件名称：adc.h
//功能概要：ADC 底层驱动构件头文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2015-05-11
//=============================================================================

#ifndef _ADC_H          //防止重复定义（开头）
#define _ADC_H

#include "common.h"     //包含公共要素头文件
//=============================================================================
//函数名称：adc_init
//功能概要：初始化 ADC 模块
//参数说明：channel：通道号
//          accurary：采样精度，单端 8、10、12
//=============================================================================
void adc_init(uint_8 channel,uint_8 accurary);
```

```
//=========================================================================
//函数名称：adc_readonce
//功能概要：进行一次 A/D 采样
//参数说明：channel：单端输入通道范围 0～31
//=========================================================================
uint_16 adc_readonce(uint_8 channel);

//=========================================================================
//函数名称：ad_mid
//函数返回：16 位无符号的 A/D 值，中值滤波后的结果(范围为 0～4095)
//参数说明：channel，通道号
//功能概要：采样三次，取中值
//=========================================================================
uint_16 adc_mid(uint_8 channel);

//=========================================================================
//函数名称：ad_ave
//函数返回：16 位无符号的 A/D 值，中值+均值滤波后的结果(范围为 0～4095)
//参数说明：channel：通道号
//         N:均值次数(范围为 0～255)，每次调用中值滤波获得
//功能概要：中值+均值滤波,实际采样次数=3*N
//=========================================================================
uint_16 adc_ave(uint_8 channel, int N);
#endif                //防止重复定义（开头）
```

2．ADC 驱动构件的使用方法

ADC 驱动构件的头文件（adc.h）中包含的内容有：给出两个对外服务函数的接口说明及声明，函数包括 ADC 初始化函数（adc_init）、读取通道数据（adc_read）。

现在，以通道 0 的 ADC 采集数据为例介绍构件的使用方法，使用举例步骤如下。

（1）初始化 5 通道，12 位精度 A/D 转换。

```
adc_init(5,12);
```

（2）读取通道 5，6×3=18 次滤波平均，赋给 16 位无符号整型变量 advalue。

```
advalue=adc_ave(5,6) ;
```

（3）在串口调试工具观察通道 5 的输出 advalue 变量。

```
printf("%u", advalue);
```

10.2 ADC 构件的制作方法

本节主要介绍如何根据 ADC 模块的各个寄存器的功能，结合上文给出的 adc.h 编写具体的 ADC 的驱动。

10.2.1　ADC 转换模块编程结构

KEA128 的 A/D 转换模块有 8 个寄存器，包括 5 个 ADC 状态控制寄存器（ADC_SC1、ADC_SC1、ADC_SC3、ADC_SC4、ADC_SC5），转换结果寄存器（ADC_R），比较值寄存器（ADC_CV）和引脚控制寄存器（ADC_APCTL1）。

1．相关名词解释

（1）**转换完成标志**：指示一个 A/D 转换是否完成，仅当 A/D 转换完成后才能从寄存器中读取数据。

（2）**通道**：ADC 模块有专门的 A/D 转换通道，分别对应着的芯片的不同引脚，读取相应引脚的数据相当于读取了通道的数据。

（3）**硬件触发**：靠外部硬件的脉冲触发。

（4）**软件触发**：软触发是靠软件编程的方式触发启动，一旦程序编写好了，触发启动是自动的、有规律的，除非修改程序，否则无法根据自己的意愿随意触发。

（5）**FIFO 队列**：用于保存采集的 A/D 数据的先进先出的队列。

2．寄存器地址分析

KEA128 模块的寄存器模块的寄存器地址如表 10-2 所示。

表 10-2　ADC 模块寄存器说明

地 址 偏 移	绝 对 地 址	寄存器名称	宽　度	访 问 方 式	重　置　值
0	4003_B000	状态控制寄存器 1（ADC_SC1）	32	R/W	0000_001Fh
1	4003_B004	状态控制寄存器 2（ADC_SC2）	32	R/W	0000_0008h
2	4003_B008	状态控制寄存器 3（ADC_SC3）	32	R/W	0000_0000h
3	4003_B00C	状态控制寄存器 4（ADC_SC4）	32	R/W	0000_0000h
5	4003_B010	转换结果寄存器（ADC_R）	32	R	0000_0000h
7	4003_B014	比较寄存器（ADC_CV）	32	R/W	0000_0000h
0	4003_B018	引脚控制寄存器（ADC_APCTL1）	32	R/W	0000_0000h
1	4003_B01C	状态控制寄存器 5（ADC_SC5）	32	R/W	0000_0000h

3．ADC 状态控制寄存器（ADC Status and Control Registers）

（1）状态控制寄存器 ADC_SC1。

数据位	D31～D8	D7	D6	D5	D4	D3	D2	D1	D0
读/写	0	COCO	AIEN	ADCO	ADCH				
复位	0				1				

D31～D8—保留，值始终为 0。

D7（COCO）—转换完成标志。COCO 标志是一个只读位，在每次转换完成，比较功能关闭（ADC_SC2[ACFE]=0）时被设置。当比较功能使能（ADC_SC2[ACFE]=1）时，COCO 标志只有在比较结果为真时被设置为转换完成。当 FIFO 功能使能（ADC_SC4[AFDEP]>0），

COCO 标志将会被设置为该组的 FIFO 转换完成。这一位在 ADC_SC1 被写或当 ADC_R 被读过之后清除。COCO=0 转换未完成；COCO=1 转换完成。

D6（AIEN）—中断使能。当 COCO 被置位且 AIEN 为高电平，产生一个中断。AIEN=0 转换完成中断关闭；AIEN=1 转换完成中断使能。

D5（ADCO）—连续转换使能。ADCO=0 选择软件触发操作，转换是发生在置 ADC_SC1 之后的;当选择硬件触发操作,转换是发生在 ADHWT 置位后。当 FIFO 功能使能(AFDEP>0)，在 ADC_SC2[ADTRG]=0 或 ADC_SC2[ADTRG]=1 且 ADC_SC4[HTRGME]=1 时，触发一组转换。ADCO=1 当选择软件触发操作时，连续转换是发生在置 ADC_SC1 之后的；当选择硬件触发操作时，转换是发生在 ADHWT 置位后。当 FIFO 功能使能（AFDEP>0），循环触发一组转换。

D4～D0（ADCH）—输入通道选择。ADCH 位形成一个 5 位字段，用来选择一个通道。ADCH=00000～01111 时分别选择 AD0～AD15；ADCH=10000～10011 选择 V_{SS}；10100～10101 保留；ADCH=10110 选择温度传感器；ADCH=10111 选择带隙；11000～11100 保留；ADCH=11101 选择 V_{REFH}；ADCH=11110 选择 V_{REFL}；ADCH=11111：模块关闭。注意：在 FIFO 模式会中重置 FIFO。

（2）状态控制寄存器 ADC_SC2。

D31～D8	D7	D6	D5	D4	D3	D2	D1	D0
0	ADACT	ADTRG	ACFE	ACFGT	FEMPTY	FFULL	REFSEL	

复位值全为 0。

D31～D8—保留，值始终为 0。

D7（ADACT）—转换活动位，指示转换正在进行。在转换完成或中止时被初始化或清除时被置位。ADACT=0 表示转换不在进行；ADACT=1 表示转换正在进行。

D6（ADTRG）—转换触发选择，选择用于启动转换触发的类型。有两种触发器可供选择：软件触发和硬件触发。当选择软件触发时，会在对 ADC_SC1 写操作后初始化转换；当选择硬件触发时，则会在 ADHWT 输入中断后初始化转换。ADTRG=0 表示软件触发；ADTRG=1 表示硬件触发。

D5（ACFE）—比较功能使能，使能比较功能。ACFE=0 比较关闭；ACFE=1 比较开启。

D4（ACFGT）—比较功能配置，当输入的转换结果大于等于比较值时触发，比较功能默认是在输入结果小于比较值时触发的。ACFGT=0 小于比较参考触发；ACFGT=1 大于等于比较参考触发。

D3（FEMPTY）—FIFO 结果是否为空。FEMPTY=0 表明 ADC 的 FIFO 至少有一个有效的新数据；FEMPTY=1 表明 ADC 的 FIFO 没有新数据。

D2（FFULL）—FIFO 结果是否满。FFULL=0 表明 ADC 的 FIFO 未满，且下一个转换数据仍可被存储进 FIFO 中；FFULL=1 表明 ADC 的 FIFO 已满，且下一个转换数据在没有读取队列操作的情况下将会覆盖老的数据

D1～D0（REFSEL）—参考电压选择。选择用于转换的参考电压源。REFSEL=00 选择默

认参考电压引脚对（V_{REFH}/V_{REFL}）；REFSEL=01 选择模拟电源引脚对（V_{DDA}/V_{SSA}）；REFSEL=10 保留；REFSEL=11 选择默认参考电压引脚对（V_{REFH}/V_{REFL}）。

（3）状态控制寄存器 ADC_SC3。

数据位	D31～D8	D7	D6	D5	D4	D3	D2	D1	D0
读/写	0	ADLPC	ADIV		ADLSMP	MODE		ADICLK	

复位值全为 0。

D31～D8—保留，值始终为 0。

D7（ADLPC）—低功耗配置。ADLPC 控制逐次逼近转换器的速度和功率的配置，在高采样率时不需要进行功耗优化。ADLPC=0 表示高速配置；ADLPC=1 表示低功耗配置。以最大时钟速度运行的是为了降低了功耗。

D6～D5（ADIV）—时钟分频选择。ADIV 选择由 ADC 使用的分频比来产生内部时钟 ADCK。ADIV=00 表示分频比为 1，时钟频率=输入时钟；ADIV=01 表示分频比为 2，时钟频率=输入时钟/2；ADIV=10 表示分频比为 3，时钟频率=输入时钟/4；ADIV=11 表示分频比为 4，时钟频率=输入时钟/8。

D4（ADLSMP）—长采样时间配置。ADLSMP 在长采样率和短采样率之间选择，这样可以通过调整采样周期来达到更高阻抗的精确采样或在低阻抗最大限度的提高采样率。如果高转换率不是必要的，长采样时间同样可以在连续的转换被启用时被用来降低整体功耗。ADLSMP=0 表示短采样时间；ADLSMP=1 表示长采样时间。

D3～D2（MODE）—转换模式选择。MODE 位表示在 12 位、10 位或 8 位操作模式之间选择。MODE=00、01、10 分别表示选择 8 位、10 位、12 位的操作模式；11 是保留位。

D1～D0（ADICLK）—输入时钟选择。ADICLK 位选择输入时钟源来产生内部时钟 ADCK。ADICLK=00 选择总线时钟；ADICLK=01 选择总线时钟/2；ADICLK=10 选择备用时钟（ALTCLK）；ADICLK=11 选择异步时钟（ADACK）。

（4）状态控制寄存器 ADC_SC4。

数据位	D31～D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
读/写	0	HTRGME	0	ASCANE	ACFSEL	0		AFDEP		

复位值全为 0。

D31～D9—保留，值始终为 0。

D8（HTRGME）—硬件触发多重转换使能。HTRGME =0 表示一个硬件触发一次转换；HTRGME=1 表示在 FIFO 模式下，一个硬件触发多重转换。

D7—保留，值始终为 0。

D6（ASCANE）—FIFO 扫描模式使能。FIFO 在使能时总是使用第一个空闲的 FIFO 通道。当这一位被设置并且 FIFO 使能时，ADC 将会重复使用第一个 FIFO 通道作为转换通道，直到 FIFO 结果完成。在连续的模式中（ADCO=1），ADC 将会在 COCO 被设置的情况下在同一个通道开始下一个转换。ASCANE=0 表示 FIFO 扫描模式关闭；ASCANE=1 表示 FIFO 扫描模式开启。

D5（ACFSEL）—比较功能选择位，在 FIFO 功能启用时，比较功能会选择 OR/AND。当这个字段被清 0 时，ADC 将会用 OR 所有的比较触发器，并且在所有比较触发器触发后设置 COCO；当这个字段被置 1 时，ADC 将会用 AND 所有的比较触发器，并且在所有比较触发器触发后设置 COCO。ACFSEL=0 表示对所有比较寄存器进行或操作；ACFSEL=1 表示对所有比较寄存器进行与操作。

D4～D3—保留，值始终为 0。

D2～D0（AFDEP）—FIFO 深度，使能 FIFO 功能并设置 FIFO 的深度。当 AFDEP 被清 0，FIFO 就关闭了。当 AFDEP 被设置为非零，FIFO 功能被使能，且深度由 AFDEP 位指定。在 FIFO 使能时，ADC_SC1[ADCH]和 ADC_R 必须由 FIFO 模式访问。在模拟通道 FIFO 的电平是由 AFDEP 指定时，ADC 开始转换。在一组转换完成并且 FIFO 结果的电平是由 AFDEP 指定时，COCO 被设置。AFDEP=000 表示 FIFO 关闭；AFDEP=001、010、011、100、101、110、111 分别表示 2～8 级深度的 FIFO 使能。

（5）ADC 状态控制寄存器 ADC_SC5。

D32～D2—保留，值始终为 0。

D1（HTRGMASKE）—硬件触发掩码使能。HTRGMASKE=0 表示硬件触发掩码除能；HTRGMASKE=1 表示硬件触发掩码使能，硬件触发不能触发 ADC 转换。

D0（HTRGMASKSEL）—硬件触发模式选择，该位选择硬件触发掩码模式。HTRGMASKSEL=0 表示根据 HTRGMASKE 选择硬件触发掩码；HTRGMASKSEL=1 表示 FIFO 非空时，自动选择硬件触发掩码。

4．ADC 转换结果寄存器（ADC Conversion Result Register）

在 12 位模式下，ADC_R 包含 12 位的 12 位转换结果。在 10 位模式下，ADC_R 包含 10 位的 10 位转换结果。在 8 位模式下，ADC_R 包含 8 位的 8 位转换结果。在每次转换完成时 ADC_R 就会被更新，除非是在使能自动比较或比较条件不满足的情况下才不会更新。当 FIFO 使能时，FIFO 结果可以通过 ADC_R 来读取。在输入通道 FIFO 满足 AFDEP 指定的深度时，ADC 转换视为完成。ADC 的 FIFO 结果可以通过 ADC_R，连续的、按照输入通道 ADCH 设定的顺序读取。如果 MODE 位改变了，在 ADC_R 中的所有数据都变成无效的。

D31～D12—保留，值始终为 0。

D11～D0（ADR）—转换结果。

5．ADC 比较值寄存器（ADC Compare Value Registers）

该寄存器保存比较值，ADCV[11：0]与一个在 12 位模式下产生的 12 位结果比较。

D31～D12—保留，值始终为 0。

D11～D0（CV）—与产生的转换结果进行比较。

6．ADC 引脚控制寄存器 1（ADC Pin Control 1 Register）

引脚控制寄存器关闭了 MCU 引脚用于模拟输入的 I/O 端口控制，用来控制与 ADC 模块 0～31 通道关联的引脚。

D31～D16—保留，值始终为 0。

D15～D0（ADPC）—ADC 引脚控制。ADPCx 控制与 ADx 通道相关联的引脚。0 表示 ADx 引脚 I/O 控制使能；1 表示 ADx 引脚 I/O 控制关闭。

10.2.2　ADC 构件封装要点和函数分析

A/D 模块具有初始化、采样、滤波等操作。按照构件的思想，可将它们封装成独立的功能函数。A/D 构件包括头文件 adc.c 和 adc.h 文件，A/D 构件头文件中主要包括相关宏定义、A/D 的功能函数原型说明等内容；A/D 构件程序文件的内容是给出 A/D 各功能函数的实现过程。

1．模块初始化（void adc_init）

初始化 ADC 模块要配置采样精度和硬件滤波以及均值滤波（KEA128 暂不支持硬件和均值滤波，故而使用软件硬件和中值滤波），首先应该先打开 ADC 模块的时钟。需要注意的是：ADC 模块共有 16 个外部通道。

2．数据读取（uint_16 adc_readonce）

初始化完成后，ADC 的各个通道就已经在进行转换了，但是在数据读取前，应该确保一次转换已将完成，所以在每次读取前，都应该判断下转换完成的标志，然后再读取数据。

10.2.3　ADC 驱动构件的实现

1．基本编程步骤

（1）首先计算出 ADC 模块及各个寄存器的基址，打开 ADC 模块时钟。

（2）配置 ADC_SC3 寄存器的总线时钟位、时钟分频位以及采样精度位；然后配置 ADC_SC2 寄存器的硬件触发位；最后配置 ADC_SC1 寄存器的 ADC 模块使能位和连续转换使能位。

配置完上述寄存器后，ADC 模块已经开始工作了，但是还需读取结果寄存器（ADC_R）才能得到具体的 A/D 转换数据。

（3）判断状态的寄存器 ADC_SC1 的 COCO 位，判断一次转换是否完成。若完成，便读取结果寄存器 ADC_R 的数据位，获取具体的数据。

2．ADC 驱动构件源程序文件（adc.c）

```
//=============================================================================
//文件名称：adc.c
//功能概要：ADC 底层驱动构件源文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2015-05-11   V1.0
//=============================================================================
#include "adc.h"
```

```
//=============================================================================
//函数名称：adc_init
//功能概要：初始化一个 A/D 转换通道
//参数说明：channel：通道号
//         accurary：单端采样精度，8、10、12
//=============================================================================

void adc_init(uint_8 channel,uint_8 accurary)
{
    //1.打开 ADC 模块时钟
    SIM_SCGC |= SIM_SCGC_ADC_MASK;
    //开启引脚的 A/D 功能
    ADC_APCTL1 |=0x0F;
    //选择总线时钟
    ADC_SC3 = (ADC_SC3 & ～ADC_SC3_ADICLK_MASK) | ADC_SC3_ADICLK(0b00);
    //输入时钟 4 分频
    ADC_SC3 = (ADC_SC3 & ～ADC_SC3_ADIV_MASK) | ADC_SC3_ADIV(0b10);
    //2.2 根据采样精度定 ADC_SC3_MODE 位
    switch(accurary)
    {
        case 8:
            ADC_SC3 |= ADC_SC3_MODE(0);      //选择 8 位转换模式
            break;
        case 10:
            ADC_SC3 |= ADC_SC3_MODE(1);      //选择 10 位转换模式
            break;
        case 12:
            ADC_SC3 |= ADC_SC3_MODE(2);      //选择 12 位转换模式
            break;
        default:
            ADC_SC3 |= ADC_SC3_MODE(2);      //选择 12 位转换模式
            break;
    }
    //3.配置 ADC_SC2：软件触发，比较功能禁用；默认外部参考电压 VREFH/VREFL
    ADC_SC2 = 0x00;
    //4.配置 ADC_SC1：使能 ADC，并设置为连续转换模式,使能外部引脚
    ADC_SC1|= ADC_SC1_ADCO_MASK;
    //选择采样通道
    ADC_SC1|=ADC_SC1_ADCH(channel);
}

//=============================================================================
//函数名称：adc_readonce
//功能概要：对 A/D 通道进行一次采样
//参数说明：channel：通道范围为 0～31
//=============================================================================
uint_16 adc_readonce(uint_8 channel)
```

```
{
    uint_16 ADCResult = 0;
    //设置 SC1 寄存器通道号
    ADC_SC1 = (ADC_SC1 & ～ADC_SC1_ADCH_MASK) | ADC_SC1_ADCH(channel);

    //等待转换完成
    while(!(ADC_SC1 & ADC_SC1_COCO_MASK));

    //读取转换结果
    ADCResult = (uint_16)ADC_R;
    //清 ADC 转换完成标志
    ADC_SC1 &= ～ADC_SC1_COCO_MASK;
    //返回读取结果
    return ADCResult;
}
//=========================================================================
//函数名称：ad_mid
//函数返回：16 位无符号的 A/D 值，中值滤波后的结果(范围为 0～4095)
//参数说明：channel：通道号
//功能概要：采样三次，取中值
//=========================================================================
uint_16 adc_mid(uint_8 channel)
{
    uint16_t i,j,k,tmp;
    //1.取 3 次 A/D 转换结果
    i = adc_readonce(channel);
    j = adc_readonce(channel);
    k = adc_readonce(channel);
    //2.取中值
    if (i > j)
    {
        tmp = i; i = j; j = tmp;
    }
    if (k > j)
        tmp = j;
    else if(k > i)
        tmp = k;
    else
        tmp = i;
    return tmp;
}

//=========================================================================
//函数名称：ad_ave
//函数返回：16 位无符号的 A/D 值，中值+均值滤波后的结果(范围为 0～4095)
//参数说明：channel：通道号
//          N:均值次数(范围为 0～255)，每次调用中值滤波获得
```

```
//功能概要：中值+均值滤波,实际采样次数=3*N
//==========================================================================
uint_16 adc_ave(uint_8 channel, int N)
{

    long int i;
    float tmp;
   int  j;

   i=0;
   for(j = 0; j < N; j++) i=i+(long int)adc_mid(channel);
   tmp =i / N;

   return (uint_16)tmp;
}
```

3．ADC 测试主函数文件 main.c

```
//说明见工程文件夹下的 Doc 文件夹内 Readme.txt 文件
#define GLOBLE_VAR      //只需在 main.c 中定义一次，用来防止全局变量的重复定义

#include "includes.h"

int main(void)
 {
   //1. 声明主函数使用的变量
   uint_32 mRuncount;                    //主循环计数器
   uint_16 ADCResult[8];                 //存放 A/D 结果
   int temp;
   int i;
   float VTemp;
   //2. 关总中断
   DISABLE_INTERRUPTS;                   //关总中断
   //3. 初始化外设模块
   uart_init (UART_Debug,9600);          //初始化"调试串口"
   light_init(LIGHT_0, LIGHT_OFF);       //初始化灯 0
   light_init(LIGHT_1, LIGHT_OFF);       //初始化灯 1
   light_init(LIGHT_2, LIGHT_OFF);       //初始化灯 2
   light_init(LIGHT_3, LIGHT_OFF);       //初始化灯 3
   adc_init(3,10);                       //初始化，通道组、单端输入，采样精度
   printf("Hello ADC! 2015-5-11\r\n");//串口发送初始化提示
   //4. main 中使用的局部变量初始化
   mRuncount = 0;                        //主循环计数器
   //5. 全局变量初始化
   //6. 使能模块中断
   uart_enable_re_int(UART_2);           //使能串口 2 接收中断
   //7. 开总中断
   ENABLE_INTERRUPTS;                    //开总中断
```

```
    //进入主循环
    //主循环开始=================================================================
    for (;;)
    {
        //运行指示灯闪烁-------------------------------------------------
        mRuncount++;                          //主循环次数计数器+1
        if (mRuncount >= RUN_COUNTER_MAX)     //主循环次数计数器大于设定的宏常数
        {
            mRuncount = 0;
            light_change(LIGHT_0);            //灯 0（RUN_LIGHT_0）状态变化
            light_change(LIGHT_1);            //灯 1（RUN_LIGHT_1）状态变化
            light_change(LIGHT_2);            //灯 2（RUN_LIGHT_2）状态变化
            light_change(LIGHT_3);            //灯 3（RUN_LIGHT_3）状态变化
            //以下加入用户程序-----------------------------------------------
            //进行一次各个通道采样
            ADCResult[0] = adc_read(0);
            ADCResult[1] = adc_read(1);
            ADCResult[2] = adc_read(2);
            ADCResult[3] = adc_read(3);
            ADCResult[4] = adc_read(4);
            ADCResult[5] = adc_read(5);
            ADCResult[6] = adc_read(6);
            ADCResult[7] = adc_read(7);
            //芯片温度采集通道
            for(i=0;i<8;i++)
            {
                  printf("%d\t",ADCResult[i]);
            }
            printf("\n");
            }
    }  //end_for

    //主循环结束=================================================================
    return 0;
}
```

4．ADC 测试工程

测试工程功能概述如下。

（1）串口通信格式：波特率 9600，1 位停止位，无校验。

（2）上电或按复位按钮时，调试串口输出“Hello ADC! 2015-5-11”。

（3）主循环中，改变 LIGHT_0～LIGHT_3 小灯状态（四盏灯同时闪烁）。调试串口输出 AD0～AD7 的 A/D 值，当配置精度为 10 位时，A/D 值范围为 0～1023；当配置精度为 12 位时，A/D 值范围为 0～4095。

（4）PC 向 MCU 发送数据时，MCU 进入串口接收中断，将接收的一个字节直接回发。三个串口具有相同功能。

ADC 构件的测试工程位于网上光盘中的“..\program\CH10_KEA128-ADC”文件夹。



10.3 KEA128 芯片 ACMP 驱动构件及使用方法

10.3.1 ACMP 编程知识要素

比较器模块可以比较模拟电压，很多场合需要检测模拟电压。例如，一个湿度报警器，传感器模拟信号经过放大后直接与比较器输入端连接，和参考电压比较，当大小发生变化时，就可以产生中断，实现可控的输出结果。比较器可用作模拟电路和数字电路的接口，还可以用作波形产生和变换电路等。利用简单电压比较器可将正弦波变为同频率的方波或矩形波。

KEA128 芯片的比较器模块（Analog Comparator，ACMP）可以比较两路模拟电压然后输出比较结果。KEA128 的 ACMP 模块的运行电源电压范围为 2.7～5.5 V；最多支持 4 个可选比较器输入（3 个外部参考和一个 DAC 输入）；可选择上升沿中断、下降沿中断或上升下降沿中断三种中断模式。其中模拟 MUX 提供电路可以从 4 路通道中选择一路模拟输入信号。

内部的 6 位 DAC 也可以提供一个电压信号，MUX 电路可以在整个电源电压范围内进行操作。6 位 DAC 是一个 64-tap 的电阻型阶梯网络，它可以为应用程序提供一个可选的参考电压。它可以将电源电压分成 64 个不同的参考电压值。每 6 位数值对应一个输出电压，它的值的范围为 Vin 到 Vin/64。

10.3.2 ACMP 引脚的标识

KEA128 共有两个 ACMP 模块 ACMP0 和 ACMP1，每个 ACMP 模块都有对应的引脚。

模块	ACMP_IN0	ACMP_IN1	ACMP_IN2	ACMP_OUT
ACMP0	PTA0	PTA1	PTC4	PTA4
ACMP1	PTA6	PTA7	PTB2	PTB5

10.3.3 KEA128 芯片 ACMP 构件头文件及使用方法

头文件 acmp.h 中的内容可分为两个主要的部分，它们分别是两个函数原型的声明和外设模块寄存器相关信息的定义。ACMP 头文件 acmp.h 含有寄存器和标志位定义，以及比较器模块相关函数声明。初始化函数 cmp_init，将参考电压选择、正向输入、负向输入通道号被设计为函数参数。设置 DAC 的值函数 dac_set_value，参数为设置 DAC 的输出比较值。这样的话，应用程序和上层构件在使用（调用）它时，将具有极大的灵活性。文件还给出了必要的硬件相关信息，当要把该构件移植到其他芯片时，就必须检查并修改这些信息。

1．ACMP 驱动构件头文件（acmp.h）

```
#include "common.h"   //包含公共要素头文件
#ifndef ACMP_H_
#define ACMP_H_
//#define REFERENCEVDD 1
#define REFERENCEVDD 0
//=========================================================================
//函数名称：cmp_init
//函数返回：无
//参数说明：reference:参考电压选择，0=Vin1in；1=Vin2in
//         plusChannel：正比较通道号
//         minusChannel：负比较通道号
//         通道号 0,1,2,3,4,5 对应引脚 PTC6,PTC7,PTC8,PTC9,PTC30,PTC29；
//         通道 6 Bandgap;通道 7 DAC 输入
//功能概要：CMP 模块初始化
//=========================================================================
void cmp_init(uint_8 reference,uint_8 plusChannel,uint_8 minusChannel);

//=========================================================================
//函数名称：dac_set_value
//函数返回：无
//参数说明：value：DAC 输出的转换值
//功能概要：设置 DAC 的值
//=========================================================================
void dac_set_value(uint_8 value);

//=========================================================================
//函数名称：cmp_enable_int
//函数返回：无
//参数说明：无
//功能概要：开比较中断
//=========================================================================
void cmp_enable_int();

//=========================================================================
//函数名称：cmp_disable_int
//函数返回：无
//参数说明：无
//功能概要：关比较中断
//=========================================================================
void cmp_disable_int();


#endif /* ACMP_H_ */
```

2．ACMP 驱动构件的使用方法

在 ACMP 驱动构件的头文件（acmp.h）中包含的内容有：给出 4 个对外服务函数的接口说明及声明，函数包括 acmp 初始化函数（cmp_init）、设置 DAC 的值函数（dac_set_value）、开比较中断函数（cmp_enable_int）和关比较中断函数（cmp_disable_int）。

现在，以比较模块引脚 PTA1 和 DAC 的模拟输出值为例，介绍构件的使用方法。使用举例步骤如下。

（1）初始化 ACMP 模块，设置参考电压为 V_{DD}，正向输入通道为 3，负向输入通道为 1。

```
cmp_init(REFERENCEVDD,3,1);          //第一个参数即表示参考电压为 VDD
```

（2）设置 DAC 的值，每次让 DAC 的值自加，用公式计算结果，具体公式为 $V_{output}=(V_{in}/64)x(DACVAL[5:0]+1)$，其中，$V_{output}$ 的范围是 $V_{in}/64$ 到 V_{in}，V_{output} 是输出值，V_{in} 是输入值。

```
dac_set_value(dac_value);            //6 位 DAC 设置输出
```

（3）使能 ACMP 模块中断。

```
cmp_enable_int();
```

（4）将 PTA1 接 V_{DD}。

（5）将 PTA1 接地；

（6）通过串口调试工具读取比较器的输出。

10.4 ACMP 构件的制作方法

本小节主要介绍如何根据 ACMP 模块的各个寄存器的功能，结合上文给出的 acmp.h 编写具体的 ACMP 的驱动构件。

10.4.1 ACMP 模块编程结构

1．相关名词解释

正向输入：比较器的正向输入值，可以是参考电压，也可以是 DAC 的值。

负向输入：比较器的负向输入值，可以是参考电压，也可以是 DAC 的值。

敏感模式：比较器对哪种类型的电压比较敏感，共有 4 种敏感模式。

2．比较器寄存器地址分析

ACMP 模块 x 的寄存器的地址=4007_3000+x×1000+n×1（x=0～1；n=0～3，n 代表寄存器号）。

3．ACMP 控制状态寄存器（ACMPx_CS）

位	D7	D6	D5	D4	D3	D2	D1	D0
读	ACE	HYST	ACF	ACIE	ADCO	ACOPE	ACMOD	
写								
复位	0							

D7（ACE）—模拟比较器使能。ACE=0 表示禁止 ACMP 模块；ACE=1 表示使能 ACMP 模块。

D6（HYST）—模拟比较器延迟选择。HYST=0 表示延迟 20 mV；HYST=1 表示延迟 30 mV。

D5（ACF）—ACMP 中断标志位，ACMP 的输出处于 ACMOD 定义的有效边沿时，可以通过硬件同步设置 ACF。向该位写 0 可以清除 ACF 位，向该位写 1 没有影响。

D4（ACIE）—ACMP 中断使能位，使能 ACMP 的 CPU 中断。ACIE=0 表示禁止 ACMP 中断；ACIE=1 表示使能 ACMP 中断。

D3（ADCO）—ACMP 输出，读取该位将会获得当前模拟比较器的输出。ADCO 复位值为 0，ACMP 禁止时（ACE=0）ADCO 位的值为 0。

D2（ACOPE）—ACMP 输出引脚使能。ACOPE 位使能引脚逻辑电平，因此输出可以设置为外部引脚。ACOPE=0 表示 ACMP 输出不能从外部引脚输出；ACOPE=1 表示 ACMP 输出能从外部引脚输出。

D1～D0（ACMOD）—中断寄存器的触发模式选择。ACMOD=00 表示 ACMP 中断发生在输出下降沿；ACMOD=01 表示 ACMP 中断发生在输出上升沿；ACMOD=10 表示 ACMP 中断发生在输出下降沿；ACMOD=11 表示 ACMP 中断发生在输出下降或上升沿。

4．ACMP 控制寄存器 0（ACMPx_C0）

位	D7	D6	D5	D4	D3	D2	D1	D0
读	0		ACPSEL		0		ACNSEL	
写								
复位	0							

D7～D6—保留，值始终为 0。

D5～D4（ACPSEL）—ACMP 正向输入选择，ACPSEL=00、01、10 分别表示外部参考 0～2；ACPSEL=11 表示 DAC 输出。

D3～D2—保留，值始终为 0。

D1～D0（ACNSEL）—ACMP 负向输入选择，ACNSEL=00、01、10 分别表示外部参考 0～2；ACNSEL=11 表示 DAC 输出。

5．ACMP 控制寄存器 1（ACMPx_C1）

位	D7	D6	D5	D4	D3	D2	D1	D0
读	DACEN	DACREF	DACVAL					
写								
复位	0							

D7（DACEN）—DAC 使能，使能 6 位 DAC 输出。DACEN=0 表示禁止 DAC；DACEN=1 表示使能 DAC。

D6（DACREF）—DAC 参考选择，DACREF=0 表示 DAC 选择带隙作为参考电压；DACREF=1 表示 DAC 选择 V_{DDA} 作为参考电压。

D5～D0（DACVAL）—DAC 输出水平选择。通过公式选择输出电压：V_{output}= (V_{in}/64)×(DACVAL[5:0]+1)，其中，V_{output} 范围为 V_{in}/64 到 V_{in}，步长为 V_{in}/64。

6．ACMP 控制寄存器 2（ACMPx_C2）

D7～D4—保留，值始终为 0。

D3～D0（ACIPE）—ACMP 输入引脚使能，该位域控制相应 ACMP 外部引脚是否能够通过模拟输入驱动。ACIPE=0 表示不允许相应的外部模拟输入；ACIPE=1 表示允许相应的外部模拟输入。

10.4.2 ACMP 构件封装要点和函数分析

ACMP 具有模块初始化、设置 DAC 的值、中断使能、中断除能等基本操作。按照构件的思想，可将它们封装成四个独立的功能函数，初始化函数完成对 ACMP 模块的工作属性的设定；进行 DAC 值的设置等。ACMP 构件头文件中主要包括相关宏定义、ACMP 的功能函数原型说明等内容。ACMP 构件程序文件的内容是给出 ACMP 各功能函数的实现过程。

1．模块初始化（cmp_init）

CMP 初始化函数，主要完成对 CMP 模块工作的参数设定，包括工作时钟、正负通道选择，参考电压选择，还有中断使能等一些基本设置。

2．设置 DAC 的值（dac_set_value）

设置 6 位 DAC 的值。具体的计算公式为 V_{output}= (V_{in}/64)×(DACVAL[5:0]+1)，其中，V_{output} 的范围是 V_{in}/64 到 V_{in}，V_{output} 是输出值，V_{in} 是输入值。

3．中断使能（cmp_enable_int）

ACMP 中断使能，注册 ACMP 的中断号（KEA128 芯片中断号为 16）。

4．中断除能（cmp_disable_int）

关闭 ACMP 中断使能。

10.4.3 ACMP 驱动构件的实现

1．基本编程方法

（1）首先计算出比较器模块以及各个寄存器的基址，打开 ACMP 模块的时钟。

（2）根据参数的正向输入和负向输入配置 ACMP_C0 寄存器的正向输入（ACPSEL）位和负向输入（ACNSEL）位；使能 DAC 输入（DACEN）位；根据参考电压参数选择 DAC 参考电压（DACREF）位（V_{DDA} 或 Bandgap）。

完成上述的寄存器操作后，ACMP 模块便开始工作了，下面开始设置 DAC 的输入值。

（3）设置 DAC 模拟输出的值，配置 ACMP_C1 寄存器，将值存入其中，用公式计算出模拟输出值。

完成 DAC 的输入后，还需要设置比较器输出结果能够产生中断。

（4）注册 ACMP 中断，中断号为 16。

（5）使能 ACMP。

上述操作完成后，便可以使 ACMP 模块工作起来了。

2．ACMP 驱动构件源程序文件（acmp.c）

```
//============================================================================
//文件名称：acmp.h
//功能概要：ACMP 底层驱动构件头文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2015-05-11  V2.0
//芯片类型：KEA128
//============================================================================
#include "acmp.h"

//============================================================================
//函数名称：cmp_init
//函数返回：无
//参数说明：reference:参考电压选择，0=Vin1in；1=Vin2in
//         plusChannel：正比较通道号
//         minusChannel：负比较通道号
//         通道号 0,1,2,3,4,5 对应引脚 PTC6,PTC7,PTC8,PTC9,PTC30,PTC29;
//         通道 6 Bandgap;通道 7 DAC 输入
//功能概要：CMP 模块初始化
//============================================================================
void cmp_init(uint_8 reference,uint_8 posInput,uint_8 negInput)
{
    ACMP_MemMapPtr acmph=ACMP0_BASE_PTR;
    if(posInput>3)
        posInput = 3;
    if(posInput<0)
        posInput = 0;
    if(negInput>3)
        negInput = 3;
    if(negInput<0)
        negInput = 0;
    //打开 ACMP 模块时钟
    SIM_SCGC |=SIM_SCGC_ACMP0_MASK;
    //根据正向通道和负向通道配置寄存器
    ACMP_C0_REG(acmph) |=ACMP_C0_ACPSEL(posInput)|ACMP_C0_ACNSEL(negInput);
    //使能 ACMP 中断
    ACMP_C1_REG(acmph) |=ACMP_C1_DACEN_MASK;
```

```
    //选择参考电压
    if(reference==0)
    {
        ACMP_C1_REG(acmph)|=ACMP_C1_DACREF_MASK;
    }
    else
    {
        //ACMP_C1_REG(acmph)&=(0<<ACMP_C1_DACREF_SHIFT);   //delet by neil li
        ACMP_C1_REG(acmph)&= ~ACMP_C1_DACREF_MASK;         //add by neil
    }
    //add by neil li 使能比较输入脚
    if(posInput!=3)
    {
        ACMP_C2_REG(acmph) |= 1<<posInput;
    }
    if(negInput!=3)
    {
        ACMP_C2_REG(acmph) |= 1<<negInput;
    }
    //选择 ACMP 敏感模式
    ACMP_CS_REG(acmph) |= ACMP_CS_ACE_MASK|ACMP_CS_ACMOD_MASK;
}

//=========================================================================
//函数名称：dac_set_value
//函数返回：无
//参数说明：value：DAC 输出的转换值
//功能概要：设置 DAC 输出的值
//=========================================================================
void dac_set_value(uint_8 value)
{
    //通过获取模块号选择比较器基址
    ACMP_MemMapPtr acmpch = ACMP0_BASE_PTR;
    //设置 DAC 的值
    ACMP_C1_REG(acmpch) |= ACMP_C1_DACVAL(value);
}

//=========================================================================
//函数名称：cmp_enable_int
//函数返回：无
//参数说明：无
//功能概要：开比较中断
//=========================================================================
void cmp_enable_int()
{
    //通过获取模块号选择比较器基址
     ACMP_MemMapPtr acmph = ACMP0_BASE_PTR;
```

```
    //开放 CMP 接收中断,上升沿下降沿均触发
     ACMP_CS_REG(acmph) |= ACMP_CS_ACIE_MASK;
     NVIC_EnableIRQ(16);
}

//========================================================================
//函数名称：cmp_disable_int
//函数返回：无
//参数说明：无
//功能概要：关比较中断
//========================================================================
void cmp_disable_int()
{
    //通过获取模块号选择比较器基址
    ACMP_MemMapPtr acmpch = ACMP0_BASE_PTR;
    //关闭 cmp 接收中断,上升沿下降沿均关闭
    ACMP_CS_REG(acmpch)&=(～ACMP_CS_ACIE_MASK);
    //关接收引脚的 IRQ 中断
    NVIC_DisableIRQ(16);
}
```

3．ACMP 中断函数文件 isr.c

```
void acmp0_isr(void)
{
    //如果是上升沿
    if ((ACMP0_CS&ACMP_CS_ACO_MASK)==ACMP_CS_ACO_MASK)
    {
        uart_send_string(UART_2, "\r\nRising edge on ACMP0\r\n");
    }
    else
    {
        uart_send_string(UART_2, "\r\nFalling edge on ACMP0\r\n");
    }
    ACMP0_CS &= ～ACMP_CS_ACF_MASK;
}
```

4．ACMP 测试主函数文件 main.c

```
//说明见工程文件夹下的 Doc 文件夹内 Readme.txt 文件
#define GLOBLE_VAR  //只需在 main.c 中定义一次，用来防止全局变量的重复定义

#include "includes.h"

int main(void)
{
    //1. 声明主函数使用的变量
    uint_32 mRuncount;                    //主循环计数器
```

```
    uint_8 dac_value;
    //2. 关总中断
    DISABLE_INTERRUPTS;                  //关总中断
    //3. 初始化外设模块
    uart_init (UART_Debug,9600);         //初始化"调试串口"
    light_init(LIGHT_0, LIGHT_OFF);      //初始化灯 0
    light_init(LIGHT_1, LIGHT_OFF);      //初始化灯 1
    light_init(LIGHT_2, LIGHT_OFF);      //初始化灯 2
    light_init(LIGHT_3, LIGHT_OFF);      //初始化灯 3
    cmp_init(REFERENCEVDD,3,1);
    printf("Hello ACMP! 2015-5-11\r\n");   //串口发送初始化提示
    //4. main 中使用的局部变量初始化
    mRuncount = 0;                       //主循环计数器
    dac_value=0;
    //5. 全局变量初始化
    //6. 使能模块中断
    uart_enable_re_int(UART_2);          //使能串口 2 接收中断
    cmp_enable_int();                    //使能 CMP0 中断和串口 1 作为通信
    //7. 开总中断
    ENABLE_INTERRUPTS;                   //开总中断

    //进入主循环
    //主循环开始=============================================================
    for (;;) {
        //运行指示灯闪烁-------------------------------------------------
        mRuncount++;                          //主循环次数计数器+1
        if (mRuncount >= RUN_COUNTER_MAX)     //主循环次数计数器大于设定的宏常数
        {
            mRuncount = 0;                    //主循环次数计数器清 0
            light_change(LIGHT_0);            //灯 0（RUN_LIGHT_0）状态变化
            light_change(LIGHT_1);            //灯 1（RUN_LIGHT_1）状态变化
            light_change(LIGHT_2);            //灯 2（RUN_LIGHT_2）状态变化
            light_change(LIGHT_3);            //灯 3（RUN_LIGHT_3）状态变化
            dac_value+=10;
            dac_set_value(dac_value);         //6 位 DAC 设置输出
            if(dac_value>60)
               dac_value=0;
        }
        //以下加入用户程序-----------------------------------------------
    }  //主循环 end_for

    //主循环结束=============================================================
    return 0;
}
```

5. ACMP 测试工程

测试工程功能概述如下。

（1）串口通信格式：波特率 9600，1 位停止位，无校验。

（2）上电时，调试串口输出“Hello ACMP! 2015-5-11”。

（3）主循环中，改变 LIGHT_0～LIGHT_3 小灯状态（四盏灯同时闪烁）

（4）程序中将 PTA1 的值与 6 位 DAC 的值进行比较,将比较信息通过串口发送出去，当用手触碰 PTA1 引脚时串口调试程序会显示“Rising edge on ACMP0”或者“Falling edge on ACMP0”。

（5）PC 向 MCU 发送数据时，MCU 进入串口接收中断，将接收的一个字节直接回发。三个串口具有相同功能。

ACMP 构件的测试工程位于网上光盘中的“..\program\CH10_KEA128-ACMP”文件夹。

第 11 章

SPI 与 I2C 模块

本章导读：本章主要阐述 KEA128 串行外设接口 SPI、集成电路互联总线 I2C 模块的编程方法。主要内容有：（1）11.1 节简要的阐述了 SPI 的编程要素，并给出 SPI 接口的引脚复用和头文件封装函数的使用方法；（2）11.2 节重点说明 SPI 模块的技术要点、寄存器的使用方法以及封装函数的实现；（3）11.3 节简要地阐述 I2C 的编程要素，并给出 SPI 接口的引脚复用和头文件封装函数的使用方法；（4）11.4 节重点说明 I2C 模块的技术要点、寄存器的使用方法及封装函数的实现。本章内容需配合实际硬件进行阅读。实际硬件只要其 MCU 型号为 KEA128 芯片即可，与具体板无关（可参考工程中的文档说明，设置与板相关性）。

本章参考资料：本章 11.1-11.2 节（SPI）参考自《KEA128 参考手册》的第 30 章，本章 11.3-11.4 节（I2C）参考自《KEA128 参考手册》的第 31 章。

11.1 SPI 模块

11.1.1 SPI 编程知识要素

1．SPI 总线特点

串行外设接口（Serial Peripheral Interface，SPI）是 Freescale 公司推出的一种同步串行通信接口，用于微处理器和外围扩展芯片之间的串行连接。SPI 一般使用 4 条线：串行时钟线 SCK、主机输入/从机输出数据线 MISO、主机输出/从机输入数据线 MOSI 和从机选择线 $\overline{SS}$。基本原理图见图 11-1。下面我们来对 SPI 模块编程要素做一个简单的阐述。

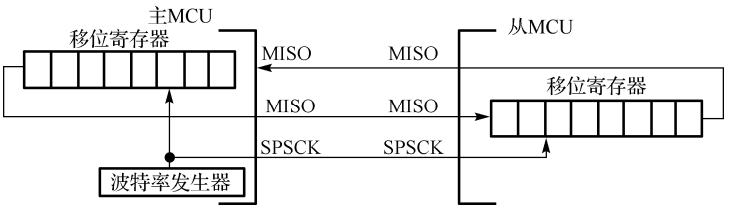


图 11-1　SPI 全双工主-从连接

一个 SPI 系统由一个主机和一个或多个从机构成，主机启动一个与从机的同步通信，从而完成数据的交换。提供 SPI 串行时钟的 SPI 设备称为 SPI 主机或主设备（Master），其他设备则称为 SPI 从机或从设备（Slave），下面来介绍 SPI 通信的基本构成。

2．从机选择引脚 $\overline{SS}$

若一个 MCU 的 SPI 工作于主机方式，则该 MCU 的$\overline{SS}$引脚设为高电平。若一个 MCU 的 SPI 工作于从机方式，当$\overline{SS}$=0 时表示主机选中了该从机，反之则未选中该从机。

3．主出从入引脚 MOSI

主出从入引脚（Master Out/Slave In，MOSI）是主机输出、从机输入数据线。对于 MCU 被设置为主机方式，主机送往从机的数据从该引脚输出。对于 MCU 被设置为从机方式，来自主机的数据从该引脚输入。

4．主入从出引脚 MISO

主入从出引脚（Master In/Slave Out，MISO）是主机输入、从机输出数据线。对于 MCU 被设置为主机方式，来自从机的数据从该引脚输入主机。对于 MCU 被设置为从机方式，送往主机的数据从该引脚输出。

5．SPI 串行时钟引脚 SCK

SPI 串行时钟引脚（Serial Clock，SCK）用于控制主机与从机之间的数据传输。串行时钟信号由主机的内部总线时钟分频获得，主机的 SCK 引脚输出给从机的 SCK 引脚，控制整个数据的传输速度。

6．时钟极性和相位详解

所谓时钟极性，是指时钟信号在空闲时是高电平还是低电平；所谓时钟相位，是指时钟信号 SCK 的第一个边沿出现在第一位数据传输周期的中央位置还是开始位置。总体要求是：确保发送数据在一周期开始的时刻上线，接收方在 1/2 周期的时刻从线上取数，这样是最稳定的通信方式。当相位 CPHA=0 时，时钟信号 SCK 的第一个边沿出现在第一位数据传输周期的中央位置，第一位数据已提前半个时钟周期送入数据线上。当相位 CPHA=1，时钟信号 SCK 的第一个边沿出现在第一位数据传输周期的开始位置。CPOL=0 在空闲时（数据发送前后）时钟信号为低电平；CPOL=1 在空闲时（数据发送前后）时钟信号为高电平。除了以上说的时钟极性和相位外还要实现主机和从机必须使用同样的时序模式，才能正常通信。

实际编程时，先要知道空闲状态情况下的时钟极性，还要知道接收方是上升沿取数还是下降沿取数。由此，确定 CPOL 及 CPHA 的值，共四种可能情况，见表 11-1。

表 11-1 确定 CPOL 及 CPHA 值的四种可能情况

选 项	空闲状态时钟极性	CPOL	接收方取数时刻	CPHA	理解方式
1	高电平	1	下降沿	0	时钟空闲为高（CPOL=1），下降沿取数，为确保数据稳定，第 1 位数据需提前上线，对应 CPHA=0。类比理解其他情况
2	高电平	1	上升沿	1	
3	低电平	0	下降沿	1	
4	低电平	0	上升沿	0	

11.1.2 KEA128 芯片 SPI 引脚

80 引脚 KEA128 芯片共有 4 组 16 个引脚可作为 SPI 引脚，其中 PTB2～5 和 PTE0～3 可复用为 SPI0 的相应的 SPI0_SCK、SPI0_MOSI、SPI0_MISO、SPI0_PCS 引脚，而剩下的 PTG4～

7 和 PTD2～4 可复用为 SPI1 的相应的 SPI0_SCK、SPI0_MOSI、SPI0_MISO、SPI0_PCS 引脚，见表 11-2。

表 11-2　KEA128 的 SPI 引脚

序号 80	引脚名	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5
1	PTD1	PTD1	KBI0_P25	FTM2_CH3	SPI1_MOSI		
2	PTD0	PTD0	KBI0_P24	FTM2_CH2	SPI1_SCK		
22	PTB5	PTB5	KBI0_P13	FTM2_CH5	SPI0_PCS	ACMP1_OUT	
23	PTB4	PTB4	KBI0_P12	FTM2_CH4	SPI0_MISO	ACMP1_IN2	NMI_b
39	PTB3	PTB3	KBI0_P11	SPI0_MOSI	FTM0_CH1	ADC0_SE7	
40	PTB2	PTB2	KBI0_P10	SPI0_SCK	FTM0_CH0	ADC0_SE6	
50	PTG7	PTG7	KBI1_P23	FTM2_CH5	SPI1_PCS		
51	PTG6	PTG6	KBI1_P22	FTM2_CH4	SPI1_MISO		
52	PTG5	PTG5	KBI1_P21	FTM2_CH3	SPI1_MOSI		
53	PTG4	PTG4	KBI1_P20	FTM2_CH2	SPI1_SCK		
57	PTD3	PTD3	KBI0_P27	SPI1_PCS			
58	PTD2	PTD2	KBI0_P26	SPI1_MISO			
67	PTE3	PTE3	KBI1_P3	SPI0_PCS			
68	PTE2	PTE2	KBI1_P2	SPI0_MISO	PWT_IN0		
75	PTE1	PTE1	KBI1_P1	SPI0_MOSI		I2C1_SCL	
76	PTE0	PTE0	KBI1_P0	SPI0_SCK	TCLK1	I2C1_SDA	

在本例程中我们将 PTE0～3 复用为 SPI0 模块作为主机端，将 PTG4～7 复用为 SPI1 模块作为从机端。连接两段对应引脚连线，实现本机两个 SPI 模块间的通信。

11.1.3　KEA128 芯片 SPI 构件头文件及使用方法

头文件实现 SPI 的初始化、SPI 的字节发送、SPI 的字节接收、SPI 的中断使能等函数。SPI 驱动构件可以实现对所有 SPI 的引脚统一编程，SPI 驱动构件由 SPI.h、SPI.c 两个文件组成，如要使用 SPI 驱动构件，只需要将这两个文件加入到所建工程中，由此方便了对 SPI 的编程操作。

1．SPI 构件头文件（SPI.h）

在 SPI 驱动构件的头文件（SPI.h）中包含的内容有：用宏定义方式定义了选择的复用引脚，包含 7 个对外服务函数的接口说明及声明。

（1）初始化：void SPI_init(uint_8 No,uint_8 MSTR,uint_16 BaudRate,uint_8 CPOL,uint_8 CPHA)。

（2）发送单个字节：uint_8 SPI_send1(uint_8 SPI_No,uint_8 data)。

（3）发送 *N* 个字节：void SPI_sendN(uint_8 SPI_No,uint_8 n,uint_8 data[])。

（4）接收单个字节：uint_8 SPI_receive1(uint_8 SPI_No)。

（5）接收 *N* 个字节：uint_8 SPI_receiveN(uint_8 SPI_No,uint_8 n,uint_8 data[])。

（6）使能接收中断：SPI_enable_re_int(uint_8 SPI_No)。

（7）禁止接收中断：SPI_disable_re_int(uint_8 SPI_No)。

```
//============================================================================
//文件名称：SPI.h
//功能概要：SPI 头文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2012-10-12   V1.0
//         2013-05-04   V2.1
//============================================================================
#ifndef SPI_H               //防止重复定义（开头）
#define SPI_H

#include "common.h"         //包含公共要素头文件
//宏定义：定义 SPI 口号
#define SPI_0  0            //SPI0 口
#define SPI_1  1            //SPI1 口
#define SPI_baseadd(SPI_nub)  (SPI_MemMapPtr)(0x40076000u+SPI_nub*0x00001000u)

#define SPI_0_GROUP 1       //SPI_0:1：PTE0～PTE3 脚，SPI_0:2：PTB2～PTB5 脚
#define SPI_1_GROUP 1       //SPI_1:1：PTG4～PTG7 脚，SPI_0:2：PTD0～PTD3 脚
//============================================================================
//函数名称：SPI_init
//功能说明：SPI 初始化
//函数参数：No：模块号，取值为 0、1
//         MSTR：SPI 主从机选择，0 选择为主机,1 选择为从机
//         BaudRate：波特率，可取 12000、6000、4000、3000、1500、1000，单位：b/s
//         CPOL：CPOL=0：高有效 SPI 时钟（低无效）；CPOL=1：低有效 SPI 时钟（高无效）
//         CPHA：CPHA=0 相位为 0；CPHA=1 相位为 1
//函数返回：无
//函数备注：CPHA=0，时钟信号的第一个边沿出现在 8 周期数据传输的第一个周期的中央；
//         CPHA=1，时钟信号的第一个边沿出现在 8 周期数据传输的第一个周期的起点。
//         CPHA=0 时，通信最稳定，即接收方在 1/2 周期的时刻从线上取数
//============================================================================
void SPI_init(uint_8 No,uint_8 MSTR,uint_16 BaudRate,uint_8 CPOL,uint_8 CPHA);

//============================================================================
//函数名称：SPI_send1
//功能说明：SPI 发送一字节数据。
//函数参数：SPI_No：模块号，其取值为 0 或 1
//         data：需要发送的一字节数据
//函数返回：0：发送失败；1：发送成功
//============================================================================
uint_8 SPI_send1(uint_8 SPI_No,uint_8 data);

//============================================================================
//函数名称：SPI_sendN
//功能说明：SPI 发送数据
//函数参数：SPI_No：模块号，其取值为 0 或 1
```

```
//          n:要发送的字节个数，范围为 1～255
//          data[]:所发数组的首地址
//函数返回：无
//=============================================================================
void SPI_sendN(uint_8 SPI_No,uint_8 n,uint_8 data[]);

//=============================================================================
//函数名称：SPI_receive1
//功能说明：SPI 接收一个字节的数据
//函数参数：SPI_No：模块号，其取值为 0 或 1
//函数返回：接收到的数据
//=============================================================================
uint_8 SPI_receive1(uint_8 SPI_No);

//=============================================================================
//函数名称：SPI_receiveN
//功能说明：SPI 接收数据，当 n=1 时，就是接受一个字节的数据
//函数参数：SPI_No：模块号，其取值为 0 或 1
//          n:要发送的字节个数，范围为 1～255
//          data[]:接收到的数据存放的首地址。
//函数返回：1：接收成功；其他情况：失败
//=============================================================================
uint_8 SPI_receiveN(uint_8 SPI_No,uint_8 n,uint_8 data[]);

//=============================================================================
//函数名称：SPI_re_enable_int
//功能说明：打开 SPI 接收中断
//函数参数：SPI_No：模块号，其取值为 0 或 1
//函数返回：无。
//=============================================================================
void SPI_enable_re_int(uint_8 SPI_No);

//=============================================================================
//函数名称：SPI_disablere_int
//功能说明：关闭 SPI 接收中断
//函数参数：SPI_No：模块号，其取值为 0 或 1
//函数返回：无。
//=============================================================================
void SPI_disablere_int(uint_8 SPI_No);

#endif   //防止重复定义（结尾）
```

2．SPI 驱动构件的使用方法

SPI 协议是通过单字节发送的，初始化参数包括：SPI 模块号、SPI 主从模式选择、SPI 波特率、SPI 的时钟极性 CPOL、SPI 的时钟相位 CPHA。CPOL、CPHA 的确定方法见表 11-1。设置好这些初始化变量后便可以开始通信，下面结合具体例子说明一下初始化的要点。首先

要通过在 spi.h 中通过宏定义来选择 SPI 的复用引脚，然后根据需要来设置主从模式，波特率和时钟极性与相位，这些参数设置完成后便完成了 SPI 的初始化。

在 SPI 驱动构件的头文件（SPI.h）中含有用宏定义方式定义 SPI 模块复用引脚，使用构件时，需要根据实际使用的引脚进行配置。

宏定义 SPI_x_GROUP（x=0～1）方便确定和改变每个 SPIx_PCS、SPIx_MOSI、SPIx_MISO、SPIx_SCK 映射到的引脚，例如，#define SPI_0_GROUP 1，表明 SPI0_PCS、SPI0_MOSI、SPI0_MISO、SPI0_SCK 引脚分别映射到引脚 PTE0、PTE1、PTE2、PTE3 上，#define SPI_1_GROUP 1，表明 SPI1_PCS、SPI1_MOSI、SPI1_MISO、SPI1_SCK 引脚分别映射到 PTG4、PTG5、PTG6、PTG7 上。

现在，以控制 SPI_0 向 SPI_1 发送数据为例，假设将 SPI0 的各个引脚复用到 PTE0～PTE3 上，将 SPI1 的各个引脚复用到 PTG4～PTG7 上，介绍构件使用方法。使用步骤举例如下。

（1）修改宏定义 SPI_0_GROUP 为 1，使 SPI_0 的复用引脚为 PTE0～PTE3，修改宏定义 SPI_1_GROUP 为 1，使 SPI_1 的复用引脚为 PTG4～PTG7。

```
//SPI_0:1：PTE0～PTE3 脚，SPI_0:2：PTB2～PTB5 脚
#define SPI_0_GROUP1                //SPI_0 使用的硬件引脚 PTE0～3
//SPI_0:1：PTG4～PTG7 脚，SPI_0:2：PTD0～PTD3 脚
#define SPI_1_GROUP1                //SPI_0 使用的硬件引脚 PTG4～7
```

（2）初始化 SPI_0 为主机模式，SPI_1 为从机模式。

```
//把 SPI0 初始化为主机,波特率 9600，时钟极性 0，时钟相位 0
SPI_init(SPI_0,1,9600,0,0);
//把 SPI1 初始化为从机,波特率 9600，时钟极性 0，时钟相位 0
SPI_init(SPI_1,0,9600,0,0);
```

（3）使能 SPI_1 接收中断，SPI_1 接收 SPI_0 模块发送的数据，并进入中断。

```
SPI_enable_re_int(SPI_1);           //使能 SPI_1 接收中断
```

（4）SPI_0 发送字符给 SPI_1。

```
SPI_send1(SPI_0,sentdata_A);        //SPI_0 发送一个字节 A。
```

（5）SPI_1 在接收中断中将数据通过串口回发到 PC 端。

```
reddata=SPI_receive1(SPI_1);        //接收主机发送过来的一个字节数据
uart_send1(UART_2, reddata);        //将接收的数据通过串口回发
```

11.1.4 SPI 主从机通信实例

1．SPI 通信实例实现

测试程序功能概述如下。

（1）SPI 通信格式：波特率 6000，SPI_0 向 SPI_1 发送数据。

（2）上电或按复位按钮时，调试串口输出“Hello SPI!”。

（3）SPI_0 向 SPI_1 发送数据，SPI_1 接收后进入接收中断，并根据接收的字符通过串口向 PC 回发数据。

测试工程见网上光盘“..\ program\CH11-KEA128-SPI”文件夹中。将 KEA128 的 PTE0～3 引脚和 PTG4～7 引脚对应连接，并把 KEA128 的串口 2 复用引脚与 PC 接好，打开串口程序，我们可以在屏幕上看到通过主机发送的数据发送到从机，接下来从机把接收到的数据再通过串口打出到屏幕上。

MCU 方的主程序及中断服务例程如下。

1）SPI 驱动构件源程序文件（main.c）

```
//说明见工程文件夹下的 Doc 文件夹内 Readme.txt 文件
#define GLOBLE_VAR  //只需在 main.c 中定义一次，用来防止全局变量的重复定义

#include "includes.h"

int main(void)
 {
   //1. 声明主函数使用的变量
   uint_32 mRuncount;       //主循环计数器
   uint_8 sentdata_A;       //定义变量，用来存储 SPI0 要发送的数据，并赋初值为 A
   uint_8 sentdata_B;       //定义变量，用来存储 SPI0 要发送的数据，并赋初值为 B
   uint_8 sentdata_Str[10];//定义变量,用来存储 SPI0 要发送的数据,并赋初值为 2015/05/06
   uint_8 flag;
   //2. 关总中断
   DISABLE_INTERRUPTS;     //关总中断
   //3. 初始化外设模块
   uart_init (UART_Debug,9600);        //初始化"调试串口"
   uart_init (UART_0,9600);
   uart_init (UART_1,9600);
   uart_init (UART_2,9600);
   light_init(LIGHT_0, LIGHT_OFF);     //初始化灯 0
   light_init(LIGHT_1, LIGHT_OFF);     //初始化灯 1
   light_init(LIGHT_2, LIGHT_OFF);     //初始化灯 2
   light_init(LIGHT_3, LIGHT_OFF);     //初始化灯 3
   //把 SPI0 初始化为主机,波特率 6000，时钟极性 0，时钟相位 0
   SPI_init(SPI_0,1,9600,0,0);
   //把 SPI1 初始化为从机,波特率 6000，时钟极性 0，时钟相位 0
   SPI_init(SPI_1,0,9600,0,0);
   printf("Hello SPI!\r\n");           //串口发送初始化提示
   //4. main 中使用的局部变量初始化
   sentdata_A='A';
   sentdata_B='B';
   strcpy(sentdata_Str,"2015/05/06");
   flag=0;
   mRuncount = 0;                      //主循环计数器
```

```
    //5. 全局变量初始化
    //6. 使能模块中断
    uart_enable_re_int(UART_Debug);     //使能 UART_Debug 的接收中断
    uart_enable_re_int(UART_0);
    uart_enable_re_int(UART_1);
    uart_enable_re_int(UART_2);
    SPI_enable_re_int(SPI_1);
    //7. 开总中断
    ENABLE_INTERRUPTS;                  //开总中断

    //进入主循环
    //主循环开始=============================================================
    for (;;)
    {
        //运行指示灯闪烁-----------------------------------------------
        mRuncount++;                            //主循环次数计数器+1
        if (mRuncount >= RUN_COUNTER_MAX)   //主循环次数计数器大于设定的宏常数
        {
            mRuncount = 0;                      //主循环次数计数器清零
            switch(flag)
            {
               case 0:
                   SPI_send1(SPI_0,sentdata_A);   //SPI_0 发送一个字节 A。
                   flag ++;
               break;
               case 1:
                   SPI_send1(SPI_0,sentdata_B);   //SPI_0 发送一个字节 B。
                   flag++;
               break;
               case 2:
                   SPI_sendN(SPI_0,10,sentdata_Str);
                   flag = 0;
               break;
            }
            light_change(LIGHT_0);                  //灯 0（LIGHT_0）状态变化
            light_change(LIGHT_1);                  //灯 1（LIGHT_1）状态变化
            light_change(LIGHT_2);                  //灯 2（LIGHT_2）状态变化
            light_change(LIGHT_3);                  //灯 3（LIGHT_3）状态变化
        }
    }
    //主循环结束=============================================================
    return 0;
}
```

2）SPI 驱动构件中断程序文件（isr.c）

```
//SPI0 接收中断服务程序
void isr_spi0_re(void)
{
```

```
    uint_8 reddata;
    DISABLE_INTERRUPTS;
    reddata=SPI_receive1(SPI_0);          //接收主机发送过来的一个字节数据。
    if(SPI_Count<2)
    {
        if(reddata == 'A')
        {
            SPI_Count++;
            printf("First\n");            //将接收到的数据发送给 PC
        }
        if(reddata == 'B')
        {
            SPI_Count++;
            printf("Second\n");           //将接收到的数据发送给 PC

        }
    }
    else
    {
        printf("2015/05/06\n");
        SPI_Count = 0;
    }
    ENABLE_INTERRUPTS;
}
```

11.1.5　SPI 构件的制作方法

1．SPI 模块的编程结构

1）KEA128 内部 SPI 模块寄存器

从编程角度来说，SPI 模块有两组模块寄存器，分别用于 SPI0 和 SPI1 模块的编程，每组主要包含 SPI_C1、SPI_C2、SPI_BR、SPI_S、SPI_D、SPI_M 六个寄存器，如表 11-3 所示。

表 11-3　SPI 模块寄存器说明

地址偏移	绝 对 地 址	寄存器名称	宽　度	访 问 方 式	重　置　值
0	4007_6000	SPI 控制寄存器 1（SPI0_C1）	8	R/W	04h
1	4007_6001	SPI 控制寄存器 2（SPI0_C2）	8	R/W	00h
2	4007_6002	SPI 波特率寄存器（SPI0_BR）	8	R/W	00h
3	4007_6003	SPI 状态寄存器（SPI0_S）	8	R	20h
5	4007_6005	SPI 数据寄存器（SPI0_D）	8	R/W	00h
7	4007_6007	SPI 匹配寄存器（SPI0_M）	8	R/W	00h
0	4007_7000	SPI 控制寄存器 1（SPI1_C1）	8	R/W	04h
1	4007_7001	SPI 控制寄存器 2（SPI1_C2）	8	R/W	00h
2	4007_7002	SPI 波特率寄存器（SPI1_BR）	8	R/W	00h
3	4007_7003	SPI 状态寄存器（SPI1_S）	8	R	20h
5	4007_7005	SPI 数据寄存器（SPI1_D）	8	R/W	00h
7	4007_7007	SPI 匹配寄存器（SPI1_M）	8	R/W	00h

SPI0、SPI1 模块的基地址分别为 4007_6000h、4007_7000h，两组基地址相差 1000h。

2）SPI 控制寄存器 1（SPIx_C1）

SPI 控制寄存器一般情况下只能复位时写一次，以后不再对其写入，不再更改对 SPI 的设置，SPIx_C1 复位后各位为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	SPIE	SPE	SPTIE	MSTR	CPOL	CPHA	SSOE	LSBFE
复位	0	0	0	0	0	1	0	0

D7—SPIE 为 SPI 中断使能（用于 SPRF 和 MODF）。这是 SPI 接收缓冲器已满（SPRF）和模式故障（MODF）事件的中断使能。SPIE=0，禁止 SPRF 和 MODF 中断；SPIE=1，当 SPRF 或 MODF 为 1 时，请求硬件中断。

D6—SPE 为 SPI 系统使能位，为 SPI 系统指定 SPI 端口引脚功能。如果 SPE 清 0，则 SPI 被禁止并强制进入空闲状态，在 S 寄存器中的所有状态位将被重置。

D5—SPTIE 为 SPI 发送中断使能，这是 SPI 发送缓冲器（SPTEF）的中断使能位。当 SPI 传送缓冲区空时产生中断。SPTIE=0，禁止 SPTEF 中断；SPTIE=1，请求硬件中断。

D4—MSTR 为主/从模式选择。MSTR=0，SPI 模块配置为从 SPI 器件；MSTR=1，SPI 模块配置为主 SPI 器件。

D3—CPOL 为时钟极性位。这个位有效地对从主 SPI 到从 SPI 器件的时钟信号串联放置一个反相器。CPOL=0，高有效 SPI 时钟（低无效）；CPOL=1，低有效 SPI 时钟（高无效）。

D2—CPHA 为时钟相位位。该位选择两种时钟格式的一种，用于不同类型的同步串行外围器件。CPHA=0，SPSCK 上的第一个边沿出现在 8 周期数据传输的第一个周期的中央；CPHA=1，SPSCK 上的第一个边沿出现在 8 周期数据传输的第一个周期的起点。

D1—SSOE 为辅选择输出使能。该位的使用结合 SPIx_C2 中的模式故障使能（MODFEN）位和主从控制位（MSTR），以确定 $\overline{SS}$ 引脚的功能，如表 11-4 所示。

表 11-4 $\overline{SS}$ 引脚的功能

MODFEN	SSOE	主 模 式	从 模 式
0	0	通用 I/O（非 SPI）	从选择输入
0	1	通用 I/O（非 SPI）	从选择输入
1	0	模式故障的 $\overline{SS}$ 输入	从选择输入
1	1	$\overline{SS}$ 从机选择输出	从选择输入

D0—LSBFE 为 LSB 优先（移位器方向）。LSBFE=0，SPI 串行数据传输以最高有效位开始；LSBFE=1，SPI 串行数据传输以最低有效位开始。

3）SPI 控制寄存器 2（SPIx_C2）

SPIx_C2 寄存器用于 SPI 系统可选项的设置，复位后各位为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	SPMIE	0	0	MODFEN	BIDIROE	0	SPISWAI	SPC0
复位	0							

D7—SPMIE 硬件匹配中断功能使能。SPME=0，禁止硬件匹配功能，即使 SPIx_M 数据

与收到的数据相同（或者说匹配）也不会触发中断；SPME=1，使能硬件匹配中断，当 SPIx_M 值与接收的数据相同时（或者说匹配时），就会触发一个中断。

D4—MODFEN 为主模式故障功能使能。当为从模式配置 SPI 时，该位没有意义或影响（SS 引脚是从选择输入）。在主模式中，该位决定 SS 引脚的使用方式（如需了解更多信息。MODFEN=0，模式故障功能禁止，主 SS 引脚恢复为不受 SPI 控制的通用 I/O；MODFEN=1，模式故障功能使能，主 SS 引脚用作模式故障输入或辅（从）选择输出。

D3—BIDIROE 为双向模式输出使能。双向模式由 SPI 引脚控制 0（SPC0=1）使能时，BIDIROE 决定 SPI 数据输出驱动器是否被使能为单个双向 SPI 的 I/O 引脚。根据 SPI 是配置为主 SPI 还是从 SPI，它将 MOSI（MOMI）或 MISO（SISO）引脚分别用作单个 SPI 数据 I/O 引脚，当 SPC0=0，BIDIROE 没有意义或影响。BIDIROE=0，输出驱动器禁止，因此 SPI 数据 I/O 引脚作为输入；BIDIROE=1，SPI I/O 引脚作为输出使能。

D1—SPISWAI 为 SPI 停止在等待模式中。SPISWAI=0，在等待模式中 SPI 时钟继续运行；SPISWAI=1，当 MCU 进入等待模式时 SPI 时钟停止。

D0—SPC0 为 SPI 引脚控制 0。SPC0 位用于选择单线双向模式。 SPC0=0，使用分开的引脚用于数据输入和数据输出（引脚模式是正常的）；SPC0=1，SPI 配置为单线双向操作 MOMI 和 SISO。

4）SPI 波特率寄存器（SPIx_BR）

该寄存器用于为一个 SPI 主机设定预定标器和位速率分频因子，它可以在任何时间被读取或写入，复位为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	0	SPPR[2:0]			SPR[3:0]			
写	—							
复位	0							

D6～D4—SPPR[2:0]为 SPI 波特率预分频系数。由这 3 位字段可以得到 SPI 波特率预分频系数，方法如下：设 SPPR[2:0]所表示的十进制数为 x，则预分频系数 SPPR=x+1。例如，SPPR[2:0]为“010”，即 x=2，由此可以得到预分频系数为 3。该预分频器的输入是总线速率时钟（BUSCLK），其输出驱动 SPI 波特率系数的输入，如图 11-2 所示。

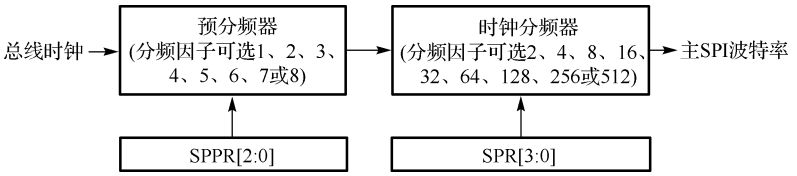


图 11-2　SPI 波特率生成

D3～D0—SPR[3:0]为 SPI 波特率系数。由这 4 位字段可以得到波特率系数，计算方法如下：设 SPR[3:0]所表示的十进制数为 y，则波特率系数 SPR=2^{y+1}。例如，SPR[3:0]=“0100”，那么 y=4，则其 SPR=2^{4+1}=32。其输入来自 SPI 波特率预分频器，输出是主模式的 SPI 波特率时钟。

通过 SPI 波特率生成示意图可以得到：SPI 主模式波特率时钟=f_{BUSCLK}/SPPR/SPR，其中 SPPR 和 SPR 可以通过上述的公式算出。

5）SPI 状态寄存器（SPIx_S）

该寄存器有四个只读状态位，位 3、2、1 和 0 没有定义，复位为 0，写入无效。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	SPRF	SPMF	SPTEF	MODF	0	0	0	0
复位	0	0	0	0	0	0	0	0

D7—SPRF 为 SPI 接收缓冲器满标志。在一次 SPI 传输完成时，SPRF 被置位，表明接收的数据可以从 SPI 数据寄存器（SPI_D）读取。当 SPRF 被置位时，通过读 SPRF，然后读取 SPI 数据寄存器，可将其清除。SPRF=0，在接收数据缓冲器中无可用数据；SPRF=1，在接收数据缓冲器中的数据可用。

D6—SPMF 位 SPI 匹配标志。SPMF=0 时，在接收数据缓冲区的值不匹配的 M 寄存器中的值；SPMF=1 时，接收数据缓冲区的值相匹配 M 寄存器中的值。要想清除该标志位，先读再写 1。

D5—SPTEF 为 SPI 发送缓冲器空标志。SPTEF=0，SPI 发送缓冲器非空；SPTEF=1，SPI 发送缓冲器空。通过读取 SPIx_S，可将其清除，向 SPIx_D 写数据之前，需清 SPTEF 位，否则写到 SPIx_D 的数据无效。在 DMA 方式下，产生 DMA 请求后，SPTEF 自动清除 0。

D4—MODF 为主模式故障标志。如果 SPI 被设置为主模式，且从模式选择输入变为低电平时，MODF 被置位，这表明某个其它的 SPI 器件也被设置成了主模式。只有 MSTR=1，MODFEN=1，且 SSOE=0 时，$\overline{SS}$ 引脚才作为模式故障错误输入；否则，MODF 将不会被置位。MODF 为 1 时，通过读 MODF，然后写入 SPI 控制寄存器 1（SPIC1），可将其清空。MODF=0，无模式故障错误；MODF=1，检测到模式故障错误。

6）SPI 数据寄存器（SPIx_D）

读取该寄存器将返回从接收数据缓冲器中读取的数据，写该寄存器将会把数据写入发送数据缓冲器。当 SPI 被配置为主模式时，写入数据到传输数据缓冲器发起一次 SPI 传输。

除非 SPI 发送缓冲器空标志（SPTEF）被置位，数据不应被写入到发送数据缓冲器，表明发送缓冲器内有空间来排列一个新的发送字节。

在 SPRF 置位后且另一传输完成之前的任意时刻，数据都可以从 SPID 中被读取。在一个新的传输完成前，从接收数据缓冲器读出数据失败，将会引起一个接收丢包状态，并且新传输的数据也会丢失。

7）匹配寄存器（SPIx_M）

寄存器存储硬件比较值，当 SPI 接收数据缓冲区收到的的值等于此硬件比较值时，SPI 匹配标志（SPMF）置位。

2．SPI 驱动构件实现

SPI 驱动构件的源程序文件中实现的对外接口函数，主要是对相关寄存器进行配置，从而完成构件的基本功能的。

```
//============================================================================
//文件名称：SPI.c
//功能概要：SPI 底层驱动构件源文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2013-05-6   V2.2
//============================================================================
#include "SPI.h"
#include "includes.h"
//============================================================================
//函数名称：SPI_init。
//功能说明：SPI 初始化
//函数参数：No：模块号，取值为 0、1
//         MSTR：SPI 主从机选择，0 选择为从机,1 选择为主机
//         BaudRate：波特率，可取 12000、6000、4000、3000、1500、1000，单位：b/s
//         CPOL：CPOL=0，高有效 SPI 时钟（低无效）；CPOL=1，低有效 SPI 时钟（高无效）
//         CPHA：CPHA=0，相位为 0；CPHA=1，相位为 1
//函数返回：无
//函数备注：CPHA=0，时钟信号的第一个边沿出现在 8 周期数据传输的第一个周期的中央
//         CPHA=1，时钟信号的第一个边沿出现在 8 周期数据传输的第一个周期的起点
//         CPHA=0 时，通信最稳定，即接收方在 1/2 周期的时刻从线上取数
//============================================================================
const uint_16 data[]={12000,0,0,6000,0,1,4000,2,0,3000,0,2,1500,0,3,
                      1000,2,2,750,1,3,500,2,3,375,1,4,125,2,5};
void SPI_init(uint_8 No,uint_8 MSTR,uint_16 BaudRate,uint_8 CPOL,uint_8 CPHA)
{
    uint_8 index;
    if(No<0||No>1)                          //如果 SPI 号参数错误则强制选择 0 号模块
    No=0;
    if(No==0)                               //对把portA的14～17引脚选择为SPI0功能。
    {
       SIM_SCGC |= SIM_SCGC_SPI0_MASK;  //SPI0 使能总线时钟
        #if(SPI_0_GROUP==1)
       //PTE0,PTE1,PTE2,PTE3 分别配置为 SPI0_CLK,SPI0_MOSI,SPI0_MISO,SPI0_PCS
       SIM_PINSEL0 |= SIM_PINSEL_SPI0PS_MASK ;
        #endif
        #if(SPI_0_GROUP==2)
       //PTB2,PTB3,PTB4,PTB5 分别配置为 SPI0_CLK,SPI0_MOSI,SPI0_MISO,SPI0_PCS
       SIM_PINSEL0 &= ～SIM_PINSEL_SPI0PS_MASK ;
        #endif
        SPI0_C1=0x00;                           //SPI 控制寄存器 1 清 0
        BSET(SPI_C1_SPE_SHIFT,SPI0_C1);     //使能 SPI 模块
        //MSTR=1 为主机模式；
        //MSTR=0 为从机模式（因 MSTR 初始值为 0，无须更改）
        (MSTR==1)?BSET(SPI_C1_MSTR_SHIFT,SPI0_C1)  \
              :BSET(SPI_C1_SPIE_SHIFT,SPI0_C1);

        //时钟极性配置，CPOL=0，平时时钟为高电平，反之 CPOL=1，平时时钟为低电平
```

```
    (0==CPOL)?BCLR(SPI_C1_CPOL_SHIFT,SPI0_C1)  \
          :BSET(SPI_C1_CPOL_SHIFT,SPI0_C1);

    //CPHA=0 时，钟信号的第一个边沿出现在 8 周期数据传输的第一个周期的中央
    //CPHA=1 时，钟信号的第一个边沿出现在 8 周期数据传输的第一个周期的起点
    (0 == CPHA)?BCLR(SPI_C1_CPHA_SHIFT,SPI0_C1)  \
           :BSET(SPI_C1_CPHA_SHIFT,SPI0_C1);

    BSET(SPI_C1_SSOE_SHIFT,SPI0_C1);    //SSOE 为 1,MODFEN 为 1,配置本模块为自动 SS 输出
    //对 SPI0 的 C1 寄存器配置为主机模式、从机选择自动模式并使能 SPI0 模块
    SPI0_C2=0x00;
   if(MSTR == 1 )                       //主机模式
      BSET(SPI_C2_MODFEN_SHIFT,SPI0_C2);
   else
     BSET(SPI_C1_SPIE_SHIFT,SPI1_C1);  //开本模块的 SPI 中断
    SPI0_BR = 0x00U;
    for(index=0;index<30;index+=3)
    {
       if(data[index]==BaudRate)
      {
          index++;
          SPI0_BR=((uint_8)data[index])<<4;//赋值给 SPI0_BR 的 SPPR 的 D6～D4
          index++;
          SPI0_BR|=((uint_8)data[index]);//赋值给 SPI0_BR 的 SPR 的 D2～D0
          break;
       }
    }

}
 else       //对把 portE 的 1～4 引脚选择为 SPI1 功能。
{
    SIM_SCGC |= SIM_SCGC_SPI1_MASK;        //SPI1 使能总线时钟
    #if(SPI_1_GROUP==1)
      //PTG4,PTG5,PTG6,PTG7 分别配置为 SPI1_CLK,SPI1_MOSI,SPI1_MISO,SPI1_PCS
      SIM_PINSEL1 |= SIM_PINSEL1_SPI1PS_MASK;
    #endif
    #if(SPI_1_GROUP==2)
     //PTD0,PTD1,PTD2,PTD3 分别配置为 SPI0_CLK,SPI0_MOSI,SPI0_MISO,SPI0_PCS
     SIM_PINSEL0 &= ～SIM_PINSEL1_SPI1PS_MASK;
    #endif

     SPI1_C1=0x00;                          //SPI 控制寄存器 1 清 0
     BSET(SPI_C1_SPE_SHIFT,SPI1_C1);     //使能 SPI 模块
     //MSTR=1 为主机模式;
     //MSTR=0 为从机模式（因 MSTR 初始值为 0，无须更改）
     (MSTR==1)?BSET(SPI_C1_MSTR_SHIFT,SPI1_C1)  \
          :BSET(SPI_C1_SPIE_SHIFT,SPI1_C1);
```

```
        //时钟极性配置，CPOL=0，平时时钟为高电平，反之 CPOL=1，平时时钟为低电平
        (0 == CPOL)?BCLR(SPI_C1_CPOL_SHIFT,SPI1_C1)  \
             :BSET(SPI_C1_CPOL_SHIFT,SPI1_C1);

        //CPHA=0 时钟信号的第一个边沿出现在 8 周期数据传输的第一个周期的中央
        //CPHA=1 时钟信号的第一个边沿出现在 8 周期数据传输的第一个周期的起点
        (0 == CPHA)?BCLR(SPI_C1_CPHA_SHIFT,SPI1_C1)  \
             :BSET(SPI_C1_CPHA_SHIFT,SPI1_C1);

        //SSOE 为 1,MODFEN 为 1，配置本模块为自动 SS 输出
        BSET(SPI_C1_SSOE_SHIFT,SPI1_C1);

        //对 SPI0 的 C1 寄存器配置为主机模式、从机选择自动模式并使能 SPI0 模块。

        SPI1_C2 = 0x00U;
        if(MSTR == 1 )                          //主机模式
            BSET(SPI_C2_MODFEN_SHIFT,SPI1_C2);
        else
           BSET(SPI_C1_SPIE_SHIFT,SPI1_C1);     //开本模块的 SPI 中断
       SPI1_BR = 0x00U;
       for(index=0;index<30;index+=3)
       {
          if(data[index]==BaudRate)
          {
              index++;
              SPI1_BR=((uint_8)data[index])<<4;//赋值给 SPI0_BR 的 SPPR 的 D6～D4
              index++;
              SPI1_BR|=((uint_8)data[index]);//赋值给 SPI0_BR 的 SPR 的 D2～D0
              break;
          }
       }
    }
}

//=========================================================================
//函数名称：SPI_send1
//功能说明：SPI 发送一字节数据。
//函数参数：SPI_No：模块号，其取值为 0 或 1
//         data：需要发送的一字节数据。
//函数返回：0：发送失败；1：发送成功。
//=========================================================================
uint_8 SPI_send1(uint_8 SPI_No,uint_8 data)
{
    uint_32 i;
    SPI_MemMapPtr baseadd=SPI_baseadd(SPI_No);
    while((!SPI_S_REG(baseadd))&SPI_S_SPTEF_MASK);//等待发送缓冲区空闲
```

```
    SPI_D_REG(baseadd)=data;
    for(i=0;i<0xFFF0;i++)      //在一定时间内发送不能完成，则认为发送失败。
    {
        if(SPI_S_REG(baseadd)&SPI_S_SPTEF_MASK)
        {
            return(1);
        }
     }
    return(0);
}

//=========================================================================
//函数名称：SPI_sendN
//功能说明：SPI 发送数据
//函数参数：SPI_No：模块号，其取值为 0 或 1
//         n:要发送的字节个数，范围为 1～255
//         data[]:所发数组的首地址
//函数返回：无
//=========================================================================
void SPI_sendN(uint_8 SPI_No,uint_8 n,uint_8 data[])
{
    SPI_MemMapPtr baseadd=SPI_baseadd(SPI_No);
    uint_32 k;
    for(k=0;k<n;k++)
    {
        while((!SPI_S_REG(baseadd))&SPI_S_SPTEF_MASK);
            SPI_D_REG(baseadd)=data[k];
    }
}

//=========================================================================
//函数名称：SPI_receive1
//功能说明：SPI 接收一个字节的数据
//函数参数：SPI_No：模块号，其取值为 0 或 1
//函数返回：接收到的数据
//=========================================================================
uint_8 SPI_receive1(uint_8 SPI_No)
{
    SPI_MemMapPtr baseadd=SPI_baseadd(SPI_No);
    while(!(SPI_S_REG(baseadd)& SPI_S_SPRF_MASK));   //检测 SPI1 是否收到数据
        return SPI_D_REG(baseadd);
}

//=========================================================================
//函数名称：SPI_receiveN
//功能说明：SPI 接收数据，当 n=1 时，就是接收一个字节的数据…
//函数参数：SPI_No：模块号，其取值为 0 或 1
//         n:要发送的字节个数，范围为 1～255
//         data[]:接收到的数据存放的首地址
```

```
//函数返回：1：接收成功,其他情况：失败。
//=============================================================================
uint_8 SPI_receiveN(uint_8 SPI_No,uint_8 n,uint_8 data[])
{
    SPI_MemMapPtr baseadd=SPI_baseadd(SPI_No);
    uint_32 m=0;
    while(m<n)
    {
        if(SPI_S_REG(baseadd)&SPI_S_SPRF_MASK)
        {
            data[m]=SPI_D_REG(baseadd);
            m++;
        }
    }
    return(1);
}

//=============================================================================
//函数名称：SPI_enable_re_int
//功能说明：打开 SPI 接收中断
//函数参数：SPI_No：模块号，其取值为 0 或 1
//函数返回：无
//=============================================================================
void SPI_enable_re_int(uint_8 SPI_No)
{
    NVIC_EnableIRQ(SPI_No+10);
}

//=============================================================================
//函数名称：SPI_disablere_int
//功能说明：关闭 SPI 接收中断
//函数参数：SPI_No：模块号，其取值为 0 或 1
//函数返回：无
//=============================================================================
void SPI_disablere_int(uint_8 SPI_No)
{
    NVIC_DisableIRQ(SPI_No+10);
}
```



11.2　I2C 模块

11.2.1　I2C 编程知识要素

1．I2C 总线特点

I2C 总线（Inter-Integrated Circuit，I2C）是二线制通信，因此只需要连接各设备的串行数据线 SDA 线和串行时钟线 SCL 线，即可实现通信，非常方便。此外，I2C 总线还支持多

主控，如果两个或更多主机同时初始化数据传输，可以通过冲突检测和仲裁防止数据被破坏。

2．I2C 总线的典型电路

图 11-3 给出一个由 MCU 作为主机，通过 I2C 总线带 3 个从机的单主机 I2C 总线系统，这是最常用、最典型的 I2C 总线连接方式。

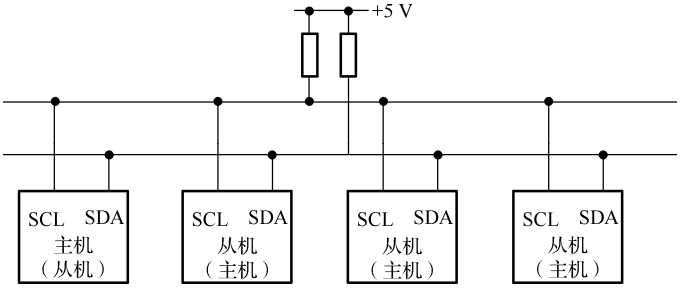


图 11-3 I2C 的典型连接

在物理结构上，I2C 系统由一条串行数据线 SDA 和一条串行时钟线 SCL 组成。主机按一定的通信协议向从机寻址并进行信息传输。在数据传输时，由主机初始化一次数据传输，主机使数据在 SDA 线上传输的同时还通过 SCL 线传输时钟。信息传输的对象和方向，以及信息传输的开始和终止均由主机决定。

每个器件都有唯一的地址存储在 A1 寄存器中，且可以是单接收的器件（如 LCD 驱动器），或者是可以接收也可以发送的器件（如存储器）。发送器或接收器可在主或从模式下操作。

3．I2C 总线信号类型分析

I2C 总线在传送数据过程中共有四种类型信号，分别是开始信号、停止信号、重新开始信号和应答信号。

1）I2C 开始信号

开始信号（START）：当 SCL 为高电平时，SDA 由高电平向低电平跳变，产生开始信号。当总线空闲的时候（例如没有主动设备在使用总线，即 SDA 和 SCL 都处于高电平），主机通过发送开始信号（START）建立通信。

2）I2C 停止信号

停止信号（STOP）：当 SCL 为高电平时，SDA 由低电平向高电平的跳变，产生停止信号。主机通过发送停止信号，结束时钟信号和数据通信。SDA 和 SCL 都将被复位为高电平状态。

3）I2C 重新开始信号

重新开始信号（Repeated START）：在 I2C 总线上，主机可以在调用一个没有产生 STOP 信号的命令后，产生一个开始信号。主机通过使用一个重复开始信号来和另一个从机通信或者同一个从机的不同模式通信。由主机发送一个开始信号启动一次通信后，在首次发送停止信号之前，主机通过发送重新开始信号，可以转换与当前从机的通信模式，或是切换到与另一个从机通信。当 SCL 为高电平时，SDA 由高电平向低电平跳变，产生重新开始信号，它的本质就是一个开始信号。

4）I2C 应答信号

应答信号（A）：接收数据的 IC 在接收到 8 位数据后，向发送数据的主机 IC 发出的特定的低电平脉冲。每一个数据字节后面都要跟一位应答信号，表示已收到数据。应答信号是在发送了 8 个数据位后，第 9 个时钟周期出现的，这时发送器必须在这一时钟位上释放数据线，由接收设备拉低 SDA 电平来产生应答信号，或者由接收设备保持 SDA 的高电平来产生非应答信号，所以一个完整的字节数据传输需要 9 个时钟脉冲。如果从机作为接收方向主机发送非应答信号，这样主机方就认为此次数据传输失败；如果是主机作为接收方，在从机发送器发送完一个字节数据后，发送了非应答信号表示数据传输结束，并释放 SDA 线。不论是以上哪种情况都会终止数据传输，这时主机或者产生停止信号释放总线，或者产生重新开始信号，从而开始一次新的通信。

开始、重新开始和停止信号都是由主控制器产生的，应答信号由接收器产生，总线上带有 I2C 总线接口的器件很容易检测到这些信号。一般情况下，一个标准的 I2C 通信由四部分组成：开始信号、从机地址传输、数据传输和结束信号。由主机发送一个开始信号，启动一次 I2C 通信，主机对从机寻址，然后在总线上传输数据。I2C 总线上传送的每一个字节均为 8 位，首先发送的数据位为最高位，每传送一个字节后都必须跟随一个应答位，每次通信的数据字节数是没有限制的；在全部数据传送结束后，由主机发送停止信号，结束通信。

11.2.2　KEA128 芯片 I2C 引脚的标识

80 引脚 KEA128 芯片共有 5 组 10 个引脚可配置为 I2C 引脚，其中 PTA2～3 和 PTB6～7 可复用为 I2C0 的相应的 I2C0_SDA、I2C0_SCL 引脚，而的 PTE0～1 和 PTH3～4 可复用为 I2C1 的相应的 I2C1_SDA、I2C1_SCL 引脚，剩下的 PTA0～1 可复用为 I2C04WSDAOUT、I2C04WSCLOUT 引脚实现 I2C 通信，见表 11-5。

表 11-5　I2C 模块实际使用的引脚

引脚号	引脚名	ALT2	ALT3	ALT4	ALT5	ALT6	TRK-KEA128 默认使用
13	PTB7	I2C0_SCL				EXTAL	
14	PTB6	I2C0_SDA				XTAL	
33	PTH4	I2C1_SCL					
34	PTH3	I2C1_SDA					
59	PTA3	UART0_TX	I2C0_SCL				I2C0_SCL
60	PTA2	UART0_RX	I2C0_SDA				I2C0_SDA
61	PTA1	FTM0_CH1	I2C0_ 4WSDAOUT	ACMP0_IN1	ADC0_SE1		
62	PTA0	FTM0_CH0	I2C0_ 4WSCLOUT	ACMP0_IN0	ADC0_SE0		
75	PTE1	SPI0_MOSI		I2C1_SCL			I2C1_SCL
76	PTE0	SPI0_SCK	TCLK1	I2C1_SDA			I2C1_SDA

在本例程中，我们将 PTA2～3 复用为 I2C0 模块作为主机端，将 PTE0～1 复用为 I2C1 模块作为从机端。连接两端对应引脚连线，实现本机两个 I2C 模块间的通信。

11.2.3 KEA128 芯片 I2C 构件头文件及使用方法

头文件实现 I2C 的初始化、I2C 的向从机的字节发送、I2C 的字节读取、I2C 的中断使能等函数。I2C 驱动构件可以实现对所有 I2C 的引脚统一编程，I2C 驱动构件由 i2c.h、i2c.c 两个文件组成，如要使用 I2C 驱动构件，只需要将这两个文件加入到所建工程中，由此方便了对 I2C 的编程操作。

1．I2C 构件头文件（i2c.h）

在 I2C 驱动构件的头文件（i2c.h）中包含的内容有：用宏定义方式定义了选择的复用引脚，包含 7 个对外服务函数的接口说明及声明。

（1）初始化函数：void i2c_init(uint_8 I2C_No, uint_8 I2C_Mode，uint_8 address,uint_8 BaudRate)。

（2）发送一字节数据：uint_8 i2c_write1(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 Access Addr, uint_8 Data)。

（3）发送 *N* 字节数据：uint_8 i2c_writen(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 Access Addr,uint_8 Data[], uint_8 N)。

（4）接收一字节数据：uint_8 i2c_read1(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 Access Addr, uint_8 *Data)。

（5）接收 *N* 字节数据：uint_8 i2c_readn(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 Access Addr, uint_8 Data[], uint_8 N)。

（6）使能 I2C 中断：void i2c_enable_re_int(uint_8 I2C_No)。

（7）关闭 I2C 中断：void i2c_disable_re_int(uint_8 I2C_No)。

```
//============================================================================
//文件名称：i2c.h
//功能概要：i2c 底层驱动构件头文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2013-03-12   V1.2
//         2015-01-21   V3.0
//============================================================================
#ifndef _I2C_H        //防止重复定义（开头）
#define _I2C_H
#include "common.h"   //包含公共要素头文件
//模块宏定义
//#define I2C0 I2C0_BASE_PTR
//#define I2C1 I2C1_BASE_PTR

//IIC 号的宏定义
#define IIC_0       0    //I2C0
#define IIC_1       1    //I2C1
```

```
//根据串口实际硬件引脚，确定以下宏常量值
//在此工程中，我们只使用 IIC0 组中的第 4 个，IIC1 组中的第 3 个，
//因此我们在此只需要将 IIC_0_GROUP 宏定义为 4,IIC_1_GROUP 宏定义为 3

//IIC_0：1=PTE24～25 脚，2=PTB0～1 脚，3=PTB2～3 脚，4=PTC8～9 脚
#define IIC_0_GROUP 1

//IIC_1: 1=PTE1～0 脚，2=PTA3～4 脚，3=PTC1～2 脚，4=PTC10～11 脚
#define IIC_1_GROUP 1


//功能接口(I2C 通信函数声明)
//============================================================================
//函数名称：i2c_init
//功能概要：初始化 IICX 模块
//参数说明：I2C_No:模块号,其取值为 0,1
//          Mode：模式，1：主机；0：从机
//          address：本模块初始化地址范围为 1～255
//          BaudRate：为波特率，其单位为 bps，其取值为 50,75,100,150,300
//函数返回：无
//============================================================================
void i2c_init(uint_8 I2C_No,uint_8 Mode,uint_8 address,uint_8 BaudRate );
//============================================================================
//函数名称：i2c_read1
//功能概要：从从机读 1 个字节数据
//参数说明：I2C_No:模块号，其取值为 0，1
//          DeviceAddr:设备地址范围为 1～255
//          AccessAddr:访问地址范围为 0～255
//          Data:带回收到的一个字节数据
//函数返回：为 0,成功读一个字节;为 1,读一个字节失败
//函数说明：内部调用 send_signal,wait
//============================================================================
uint_8 i2c_read1(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 AccessAddr, uint_8 *Data);
//============================================================================
//函数名称：i2c_write1
//功能概要：向从机写 1 个字节数据
//参数说明：I2C_No:模块号，其取值为 0，1
//          DeviceAddr:设备地址范围为 1～255
//          AccessAddr:访问地址范围为 0～255
//          Data:要发给从机的 1 个字节数据
//函数返回：为 0,成功写一个字节;为 1,写一个字节失败
//函数说明：内部调用 send_signal,wait
//============================================================================

uint_8 i2c_write1(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 AccessAddr, uint_8 Data);
//============================================================================
//函数名称：i2c_readn
```

```
//功能概要：从从机读 N 个字节数据
//参数说明：I2C_No:模块号，其取值为 0，1
//         DeviceAddr:设备地址范围为 1～255
//         AccessAddr:访问地址范围为 0～255
//         Data:读出数据的缓冲区
//         N:从从机读的字节个数
//函数返回：为 0,成功读 N 个字节;为 1,读 N 个字节失败
//函数说明：内部调用 i2c_read1
//=========================================================================
uint_8 i2c_readn(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 AccessAddr, uint_8
                 Data[], uint_8 N);
//=========================================================================
//函数名称：i2c_writen
//功能概要：向从机写 N 个字节数据
//参数说明：I2C_No:模块号，其取值为 0，1
//         DeviceAddr:设备地址范围为 1～255
//         AccessAddr:访问地址范围为 0～255
//         Data:要写入的数据的首地址
//         N:从从机读的字节个数
//函数返回：为 0,成功写 N 个字节;为 1,写 N 个字节失败
//函数说明：内部调用 i2c_write1
//=========================================================================
uint_8 i2c_writen(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 AccessAddr,uint_8
                  Data[], uint_8 N);
//=========================================================================
//函数名称：i2c_re_enable_int
//功能说明：打开 i2c 的 IRQ 中断
//函数参数：i2cNO:i2c 模块号，其取值为 0，1
//函数返回：无
//=========================================================================
void i2c_enable_re_int(uint_8 I2C_No);
//=========================================================================
//函数名称：i2c_re_disable_int
//功能说明：关闭 i2c 的 IRQ 中断
//函数参数：i2cNO:i2c 模块号，其取值为 0,1
//函数返回：无
//=========================================================================
void i2c_disable_re_int(uint_8 I2C_No);
//=========================================================================
//函数名称：send_signal
//功能概要：根据需要产生开始或停止信号
//参数说明：I2C_No:模块号，其取值为 0，1
//         Signal: 'S'(Start),产生开始信号   'O'(Over),  产生停止信号
//函数返回：无
//=========================================================================
void send_signal(uint_8 Signal,uint_8 I2C_No);
//=========================================================================
```

```
//函数名称：  wait
//功能概要：在时限内,循环检测接收应答标志位,或传送完成标志位,判断 MCU
//          是否接收到应答信号或一个字节是否已在总线上传送完毕
//参数说明：I2C_No:模块号，其取值为 0，1
//          x:x = 'A'(Ack),等待应答;x = 'T'(Transmission),等待一个字节数据传输完成
//函数返回：0:收到应答信号或一个字节传送完毕;
//          1:未收到应答信号或一个字节没传送完
//=========================================================================
uint_8 wait(uint_8 x,uint_8 I2C_No);
#endif  //防止重复定义（结尾）
```

2．I2C 驱动构件的使用方法

I2C 协议是通过单字节发送的，初始化参数包括：I2C 模块号、I2C 主从模式选择、I2C 模块地址、I2C 波特率。设置好这些初始化变量后我们便可以开始通信，下面我们结合具体例子说明一下初始化的要点，首先要通过在 i2c.h 中通过宏定义来选择 I2C 的复用引脚，然后根据需要来设置主从模式、本机地址和波特率，这些参数设置完成后我们便完成了 I2C 的初始化。

在 I2C 驱动构件的头文件（i2c.h）中含有用宏定义方式定义 I2C 模块复用引脚，使用构件时，需要根据实际使用的引脚进行配置。

宏定义 IIC_x_GROUP（x=0～1）方便确定和改变每个 IICx_SCL、IICx_SDA 映射到的引脚。例如，“#define IIC_0_GROUP 1”表明 IICx_SCL、IICx_SDA 引脚分别映射到引脚 PTE24、PTE25 上；“#define IIC_1_GROUP 1”表明 IICx_SCL、IICx_SDA 引脚分别映射到 PTE0、PTE1 上。

现在，以控制 IIC_0 向 IIC_1 发送数据为例，假设将 IIC_0 的各个引脚复用到 PTE24、PTE25 上，将 IIC_1 的各个引脚复用到 PTE0、PTE1 上，介绍构件使用方法。使用步骤举例如下。

（1）修改宏定义 IIC_0_GROUP，确定 IIC_0 的 SCL 和 SDA 复用引脚为 PTE24、PTE25 脚，IIC_1_GROUP，确定 IIC_1 的 SCL 和 SDA 复用引脚为 PTE0、PTE1 脚。

```
//IIC_0：1=PTE24～25 脚，2=PTB0～1 脚，3=PTB2～3 脚，4=PTC8～9 脚
#define IIC_0_GROUP 1    //SD-FSL-KL25-EVB 板上使用 PTC8～9 脚
//IIC_1: 1=PTE1～0 脚，2=PTA3～4 脚，3=PTC1～2 脚，4=PTC10～11 脚
#define IIC_1_GROUP 1    //SD-FSL-KL25-EVB 板上使用 PTC1～2 脚
```

（2）初始化 IIC_0 为主机模式，IIC_1 为从机模式。

```
i2c_init(IIC_1,0, SlaveAddress,100);   //i2c1 模块初始化,设置从机地址
i2c_init(IIC_0,1,MasterAddress,100);   //i2c0 模块初始化,设置主机地址
```

（3）使能 IIC_1 接收中断，IIC_1 接收 IIC_0 模块发送的数据，并进入中断。

```
i2c_enable_re_int(IIC_1);              //使能 IIC_1 接收中断
```

（4）IIC_0 发送字符给 IIC_1。

```
//主机 IIC0 向从机发送一个字节信息
i2c_write1(IIC_0, SlaveAddress, 0x02, data1[Num_flag]);
```

（5）IIC_1 在接收中断中将全局变量 flag 置位，并存入全局数组 buf 中，主函数通过判断 flag 位来发送 buf 中的数据。

```
printf("%c",buf[visitaddr]);
```

11.2.4 I2C 主从机通信实例

1．I2C 通讯实例实现

测试程序功能概述如下。

（1）上电或按复位按钮时，调试串口输出“Hello IIC!”。

（2）将 I2C0 和 I2C1 两个模块的相应的 2 个引脚复用为 I2C_SDA、I2C_SCL 功能，并进行硬件连接，对 PTA2～3 和 PTE0～1 两组 I2C 模块进行连接[①]。

（3）初始化 I2C 模块，若是主机，则设定 i2c_init 的 mode 参数为 1；若 I2C 为从机，则设置 mode 参数为 0。

（4）初始化 I2C1 为从机模式，并打开从机中断，通过配置 I2C1_C1 的来实现从机的中断。

（5）在主函数中通过调用 i2c_write1 进行 i2c 单字节传送。

（6）在 isr.c 和 isr.h 中加入 I2C 接收中断函数并实现，接收主机发送并将数据通过串口打出的函数功能。

测试工程见网上光盘“..\program\CH11-KEA128-I2C”文件夹中，测试程序主要功能是返回主机发送至从机的内容，使用 KEA128 上的 UART_2 串口连接 PC 连观察实验现象。

MCU 方的主程序及中断服务例程如下。

1）I2C 驱动构件源程序文件（main.c）

```
//说明见工程文件夹下的 Doc 文件夹内 Readme.txt 文件
#define GLOBLE_VAR  //只需在 main.c 中定义一次，用来防止全局变量的重复定义

#include "includes.h"

int main(void)
 {
    //1. 声明主函数使用的变量
    uint_32 mRuncount;                  //主循环计数器
    uint_8 data1[12];                   //发向从机的数据；
    uint_8 Num_flag;                    //data 位置指向标记
    //2. 关总中断
```

① 注意：这里的 I2C 的 SDA 信号和 SCL 信号均需要进行 5 V 上拉，上拉的方式有两种：一种方式是使用内部上拉电阻进行上拉（KEA128 当引脚复用为 I2C 功能时，并没有自动配置上拉功能，因此还需要对引脚进行上拉配置）；另一种方式是外接上拉电阻进行上拉。用户可以根据实际情况进行选择，这里为了接线方便采取内部上拉的方式，上拉的配置在 I2C 初始化函数中可以看到。

```
DISABLE_INTERRUPTS;                    //关总中断
//3. 初始化外设模块
uart_init (UART_Debug,9600);           //初始化"调试串口"
uart_init (UART_0,9600);
uart_init (UART_1,9600);
uart_init (UART_2,9600);
i2c_init(IIC_1,0,0x73,100);                //i2c1 模块初始化
i2c_init(IIC_0,1,MasterAddress,100);       //第四个参数为波特率，其单位为 kb/s
light_init(LIGHT_0, LIGHT_OFF);            //初始化灯 0
light_init(LIGHT_1, LIGHT_OFF);            //初始化灯 1
light_init(LIGHT_2, LIGHT_OFF);            //初始化灯 2
light_init(LIGHT_3, LIGHT_OFF);            //初始化灯 3
printf("Hello IIC! \r\n");                 //串口发送初始化提示
//4. main 中使用的局部变量初始化
mRuncount = 0;                             //主循环计数器
Num_flag = 0;
strcpy(data1,"Version1.0\n");              //为 data 数组赋值
//5. 全局变量初始化
flag=0;
//6. 使能模块中断
uart_enable_re_int(UART_Debug);            //使能 UART_Debug 的接收中断
uart_enable_re_int(UART_0);
uart_enable_re_int(UART_1);
uart_enable_re_int(UART_2);
i2c_enable_re_int(IIC_1);
//7. 开总中断
ENABLE_INTERRUPTS;                         //开总中断

//进入主循环
//主循环开始=============================================================
for (;;)
{
    //运行指示灯闪烁-------------------------------------------------
    mRuncount++;                           //主循环次数计数器+1
    if (mRuncount >= RUN_COUNTER_MAX)      //主循环次数计数器大于设定的宏常数
    {
        mRuncount = 0;                     //主循环次数计数器清 0
        i2c_write1(IIC_0, 0x73, 0x02, data1[Num_flag]);
        Num_flag++;
        if(Num_flag > 10 )
        Num_flag=0;
        if(flag)
           printf("%c",buf[visitaddr]);
        light_change(LIGHT_0);             //灯 0（LIGHT_0）状态变化
        light_change(LIGHT_1);             //灯 1（LIGHT_1）状态变化
        light_change(LIGHT_2);             //灯 2（LIGHT_2）状态变化
        light_change(LIGHT_3);             //灯 3（LIGHT_3）状态变化
```

```
        }
    }
    //主循环结束=============================================================
    return 0;
}
```

2）I2C 驱动构件中断程序文件（isr.c）

```
//I2C1 中断服务例程
void I2C1_isr(void)
{
    uint_8 i;
    uint_8 data;
    DISABLE_INTERRUPTS;
    if(i2c1_wait('T'))          //等待一个数据传送完成
    {
        i=I2C1_D;               //读出 IIC1D,准备接收数据
        goto lab;
    }
    //if((I2C1_S & 0x40)!=0)    //主机发送的地址与本机的默认地址匹配
    if((I2C1_S1 & I2C_S_IAAS_MASK)!=0)      //主机发送的地址与本机的默认地址匹配
    {
        i=I2C1_S1 & I2C_S_SRW_MASK;
        if(0!=j)
        {
            j=0;
            goto lab;
        }
        BCLR(I2C_C1_TX_SHIFT,I2C1_C1);      //TX = 0,MCU 设置为接收模式
        i=I2C1_D;                           //读出 IIC1D,准备接收数据
        j++;
    }
    else                                    //接收到的为数据
    {
        if(1==j)                            //接收到主机发送过来的访问地址
        {
            visitaddr=I2C1_D;               //读出 IIC1D,准备接收数据
            j++;goto lab;
        }
        if(2==j)                            //接收到主机发送过来的数据
        {
            data=I2C1_D;                    //读出 IIC1D,准备接收数据
            j++;
        }
        //如果从机地址匹配、访问地址与数据均接收到则将接收到的数据存入 buf 数组
        if(3==j)
        {
            buf[visitaddr]=data;
            flag=1;                  //置位该标志表明成功接收到主机发送过来的数据
        }
        j=0;
    }
```

```
    lab:
    ENABLE_INTERRUPTS;
}
```

11.2.5　I2C 构件的制作方法

1．I2C 模块的编程结构

从编程角度来说，I2C 模块有两组模块寄存器，分别用于 I2C0 和 I2C1 模块的编程，每组包含 5 个寄存器，见表 11-6。

表 11-6　I2C 的模块 0 和模块 1 常用的 10 个寄存器

模　块　号	寄存器名称	缩　　写	地　　址	基 本 功 能
0	地址寄存器	I2C0_A1	0x4006_6000	设置从机地址
	分频寄存器	I2C0_F	0x4006_6001	设置 I2C 模块的工作频率
	控制寄存器 1	I2C0_C1	0x4006_6002	设置传输格式、中断使能
	状态寄存器	I2C0_S	0x4006_6003	表明 I2C 模块的工作状态
	数据寄存器	I2C0_D	0x4006_6004	收发数据
1	地址寄存器	I2C1_A1	0x4006_7000	设置从机地址
	分频寄存器	I2C1_F	0x4006_7001	设置 I2C 模块的工作频率等
	控制寄存器 1	I2C1_C1	0x4006_7002	设置传输格式、中断使能等
	状态寄存器	I2C1_S	0x4006_7003	表明 I2C 模块的工作状态
	数据寄存器	I2C1_D	0x4006_7004	收发数据

I2C0、I2C1 模块的基地址分别为 4006_6000h、4006_7000h，两组基地址相差 1000h。

1）I2C 地址寄存器 1（I2Cx_A1）

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	AD[7：1]							0
写								—
复位	0							

D7～D1（AD[7：1]）当作为从机被寻址时，此字段包含 I2C 模块所使用的初始从机地址，此字段用于 7 位地址方案和 10 位地址方案中的低 7 位。

2）I2C 分频寄存器（I2Cx_F）

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	MULT		ICR					
复位	0							

D7～D6（MULT）—定义倍频因子 MUL。此因子与 SCL 分频一起使用，以产生 I2C 波特率。当 MUL 分别为 00、01、02 时，倍频因子分别为 1、2 和 4。

D5～D0（ICR）—时钟频率。此字段和 MULT 字段确定 I2C 波特率、SDA 保持时间、SCL 开始保持时间和 SCL 停止保持时间。SCL 分频因子乘以倍频因子（MUL）决定了 I2C 波特率。

I2C 波特率=总线频率（Hz）/（MUL×SCL 分频因子）

SDA 保持时间是从 I2C 时钟（SCL）的下降沿到 I2C 数据（SDA）变化的延迟时间。

SDA 保持时间=总线周期（S）×MUL×SDA 保持值

SCL 开始保持时间是当 SCL 为高电平（启动条件）时，从 SDA 的下降沿（I2C 数据）到 I2C 时钟（SCL）的下降沿的延迟。

SCL 开始保持时间=总线周期（S）×MUL×SCL 开始保持值

SCL 停止保持时间是当 SCL 为高电平（停止条件）时，从 I2C 时钟（SCL）的上升沿到 SDA 的上升沿（I2C 数据）的延迟。

SCL 停止保持时间=总线周期（s）×MUL×SCL 停止保持值

例如，如果总线频率为 8 MHz，表 11-7 给出了选取不同的 ICR 和 MULT， I2C 波特率为 100 kbps 的对应保持时间值。

表 11-7 MULT、ICR 和保持时间值

MULT	ICR	保持时间（μs）		
		SDA	SCL Start	SCL Stop
2h	00h	3.500	3.000	5.500
1h	07h	2.500	4.000	5.250
1h	0Bh	2.250	4.000	5.250
0h	14h	2.125	4.250	5.125
0h	18h	1.125	4.750	5.125

3）I2C 控制寄存器 1（I2Cx_C1）

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	IICEN	IICIE	MST	TX	TXAK	RSTA	WUEN	0
复位	0							

D7（ICEN）—I2C 使能。使能 I2C 模块的操作，0 禁用，1 使能。

D6（IICEN I 2C）—I2C 中断使能。使能 I2C 中断请求，0 禁用，1 使能。

D5（MST）—主机模式选择。当 MST 位从 0 改变为 1 时，在总线上产生一个开始信号并且主模式被选中；当该位从 1 变到 0 时，产生一个停止信号并且操作模式由主机切换到从机。0 为从机模式，1 为主机模式。

D4（TX）—传送模式选择。选择主机和从机的传输方向，在主机模式下，该位根据需要传输的类型来设置，因此，处于地址周期，该位总是置位。当作为从机被寻址时，该位必须根据状态寄存器中的 SRW 位通过软件设置。0 表示接收，1 表示发送。

D3（TXAK）—传送应答使能。在从机和主机接收器的应答周期中，指定的数据被驱动到 SDA 上。FACK 位的值影响 NACK/ACK 的产生。**注意**：SCL 保持低电平直到 TXAK 被写入，当 TXAK 为 0 时，应答信号在接收完一个字节后（如果 FACK 被清 0）或当前正在接收字节时（如果 FACK 置位）被发送到总线上；当 TXAK 为 1 时，没有应答信号在接收完一个字节后（如果 FACK 被清 0）或当前正在接收字节时（如果 FACK 置位）被发送到总线上。

D2（RSTA）—重复开始。写 1 到该位为当前的主机提供了一个产生重启的条件，该位读取值总是 0，在错误的时间尝试重启会导致仲裁丢失。

D1（WUEN）—唤醒使能。当从机地址匹配发生时，I2C 模块能够在没有外围总线运行的情况下从低功耗模式中唤醒 CPU。当 WUEN 为 0 时，正常运行，在低功耗模式下发生地址匹配时无中断产生；当 WUEN 为 1 时，在低功耗模式下使能唤醒功能。

4）I2C 状态寄存器（I2Cx_S）

数据	D7	D6	D5	D4	D3	D2	D1	D0
读	TCF	IAAS	BUSY	ARBL	RAM	SRW	IICIF	RXAK
写	—		—	W1C		—	W1C	—
复位	1	0	0	0	0	0	0	0

D7（TCF）—传输完成标志。该位在一个字节和应答信号传输完成时被置位，仅在传送到 I2C 模块或者从 I2C 模块传送出去期间或之后有效。在接收模式下读取 I2C 数据寄存器清零 TCF 位，或者在传送模式下写 I2C 数据寄存器清零它。1 正在传送，0 传送完毕。

D6（IAAS）—作为从机被寻址。满足下列条件之一时，此位被置位：①呼叫地址要与 A1 的寄存器中的可编程从机初始地址或 RA 扩展寄存器中的地址（其必须设置为非零值）相匹配；②GCAEN 被置位并且广播被接收；③SIICAEN 被置位并且广播地址匹配第二次可编程的从机地址；④ALERTEN 被置位并且 SMBus 警告响应地址被接收；⑤RMEN 被置位并且地址被接收（其值的范围在 A1 和 RA 寄存器之间）。

该位在 ACK 位之前被设置，CPU 必须检查 SRW 位并设置相应的 TX/RX，写 C1 寄存器清 0 该位。当 IAAS 为 0 时，没有被寻址；当 IAAS 为 1 时，作为从机被寻址。

D5（BUSY）—总线忙。无论主机从机模式，该位总表示总线的状态。检测到 START 号时该位被置位，检测到 STOP 信号时，该位被清 0。0 表示总线空闲，1 表示总线忙。

D4（ARBL）—仲裁丢失。当仲裁程序丢失时，此位由硬件置位。ARBL 位必须由软件写 1 来清 0。当 ARBL 为 0 时，为标准总线操作；当 ARBL 为 1 时，为仲裁丢失。

D3（RAM）—扩展地址匹配。满足下列的任何条件时此位被设置为 1：①接收任何与 RA 寄存器中地址相匹配的非零的广播地址；②RMEN 位被置位并且广播地址的值在 A1 和 RA 寄存器的值的范围内。**注意：**为了 RAM 位能被正确地设置为 1，C 1[IICIE]必须设置为 1。写 C1 寄存器清除该位为 0。当 RAM 为 0 时没有被寻址，当 RAM 为 1 时作为从机被寻址。

D2（SRW）—从机读/写。当作为一个从机时，SRW 显示发送到主机的广播地址读/写命令位的值。当 SRW 为 0 时从机接收，主机写入到从机；当 SRW 为 1 时从机发送，主机读取从机。

D1（IICIF）—中断标志。当中断被挂起时该位置位。例如，在中断程序中，该位必须通过软件写 1 来进行清 0。以下事件可以置位该位：①在接收模式下通过写 0 或者写 1 设置该位后，如果 FACK 是 0，那么一个字节（包括 ACK/NACK 位）传输完成后，一个 ACK 或 NACK 被发送到总线上；②如果 FACK 是 1，一个字节传输完成（不包括 ACK/NACK 位）；③从机地址匹配广播地址（包括初始从机地址、扩展从机地址、警告响应地址、第二个从机地址或广播）；④仲裁丢失；⑤在 SMBus 模式下，任何超时（除了 SCL 和 SDA 高超时外）；⑥如果 STOPIE 位在输入干扰滤波器寄存器中为 1，I2C 总线停止检测。注意：为了清除 I2C 总线停止检测中断，在中断服务例程中，首先通过写 1 清 0 输入电子脉冲滤波寄存器中的

STOPF 位，然后清除 IICIF 位。如果此序列反转，IICIF 位再次被声明。当 IICIF 为 0 时没有中断挂起，当 IICIF 为 1 时有中断挂起。

D0（RXAK）—接收应答。当 RXAK 为 0 时在总线上的一个字节的数据传输完成后，应答信号被接收；当 RXAK 为 1 时没有应答信号被检测到。

5）I2C 数据 I/O 寄存器（I2Cx_D）

在主机传送模式，当数据被写入到这个寄存器后，数据传输开始，首先发送最高位，在主机接收模式下，读取该寄存器开始接收下一个字节的数据。**注意：**要改变主机接收模式的话，那么在读数据寄存器之前必须改变 I2C 模式以防止不慎启动主机，把接收的数据传输出去。在从机模式下，同样的功能都可以在地址匹配后发生后获得。在主模式和从模式下的传输开始时，C1 [TX]位必须正确地反映所需的传输方向。例如，如果 I2C 模块配置为主机传送，但是却需要主机接收，那么读取数据寄存器将不会启动接收。当 I2C 模块在主机接收或从机接收模式被配置，那么读数据寄存器返回最后一个接收到的字节。数据寄存器不反映 I2C 总线上传输的每个字节，也不可以通过读回它来软件验证一个字节是否被正确写入到数据寄存器。在主机传送模式下，写入到数据寄存器中的数据的第一个字节跟随 MST（起始位）或 RSTA 的声明（重复开始位）被用于地址传输并且必须由广播地址（7～1 位）和所需的读/写位（0 位）连接组成。

2．I2C 驱动构件实现

I2C 驱动构件的源程序文件中实现的对外接口函数，主要是对相关寄存器进行配置，从而完成构件的基本功能。

```
//=============================================================================
//文件名称：i2c.c
//功能概要：I2C 底层驱动构件源文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2013-05-4   V2.1
//         2015-01-21   V3.0
//=============================================================================
#include  "i2c.h"
#include "string.h"
static const uint_16 data[]={300,0x14,150,0x54,100,0x1f,75,0x25,50,0x5f};
static void send_signal(uint_8 Signal,uint_8 I2C_No);
static uint_8 wait(uint_8 x,uint_8 I2C_No);
static I2C_MemMapPtr i2c_get_base_address(uint_8 I2C_No);
uint_8 data1[12]="Version3.4\n";
static uint_8 Num_flag=0;
//=============================================================================
//函数名称：i2c_init
//功能概要：初始化 IICX 模块。
//参数说明：I2C_No:模块号,其取值为 0,1
//         Mode：模式，1：主机；0：从机
//         address：本模块初始化地址范围为 1～255
//         BaudRate：为波特率，其单位为 bps，其取值为 50,75,100,150,300
```

```
//函数返回：无
//============================================================================
void i2c_init(uint_8 I2C_No,uint_8 Mode,uint_8 address,uint_8 BaudRate )
{
    //获取 I2C 模块的基址
    uint_8 index;
    I2C_MemMapPtr num = i2c_get_base_address(I2C_No);
    if(I2C_No<0||I2C_No>1)                          //如果模块号错误则强制其为 0
    {
        I2C_No=0;
    }
    if(num==I2C0)
    {
        //I2C0 Clock Gate Control --enable
        SIM_SCGC |= SIM_SCGC_I2C0_MASK;
        I2C0_C1=0X00;
        BSET(I2C_S_IICIF_SHIFT,I2C0_S1);
       #if(IIC_0_GROUP==1)
       //PTA2，PTA3 设置为 I2C0_SDA,I2C0_SCL
       SIM_PINSEL0 &= ～SIM_PINSEL_I2C0PS_MASK;
       PORT_PUE0 |= 0xc;
       #endif
       #if(IIC_0_GROUP==2)
       //PTB6,PTB7 设置为 I2C0_SDA,I2C0_SCL 并进行上拉
       SIM_PINSEL0 |= SIM_PINSEL_I2C0PS_MASK;
       PORT_PUE0 |= 0xc000;
       #endif
        //设置 MULT 和 ICR,MCU 总线频率为 24 MHz，在总线上分频得 75 kbps 波特率
        I2C0_A1=address;                            //本机作为从机时的默认地址
        for(index=0;index<10;index+=2)
        {
            if(data[index]==BaudRate)
            {
                index++;
                I2C0_F=data[index];                 //将定义的参数赋给 I2C1_F
                break;
            }
        }
        if(1 == Mode)                               //主机模式
        {
            BSET(I2C_C1_IICEN_SHIFT,I2C0_C1);       //开 i2c0 模块使能
            BSET(I2C_C1_TXAK_SHIFT,I2C0_C1);        //置位 i2c0 TXAK
        }
        else                                        //从机模式
        {
            BSET(I2C_C1_IICEN_SHIFT,I2C0_C1);       //使能 I2C 模块
            BSET(I2C_C1_IICIE_SHIFT,I2C0_C1);       //来 I2C 中断
```

```
            BSET(I2C_C1_MST_SHIFT,I2C0_C1);      //设置成主机模式
            BCLR(I2C_C1_TX_SHIFT,I2C0_C1);       //TX = 0,MCU 设置为接收模式
            //i=I2C0_D;                          //读出 IIC1D,准备接收数据
            BCLR(I2C_C1_MST_SHIFT,I2C0_C1);      //MST 位由 1 变成 0，设置为从机模式
        }

    }
    else
    {
        //I2C1 Clock Gate Control --enable
        SIM_SCGC |= SIM_SCGC_I2C1_MASK;
        I2C1_C1=0X00;
        BSET(I2C_S_IICIF_SHIFT,I2C1_S1);
        #if(IIC_1_GROUP==1)
        //分别将 PTE0、PTE1 设置为 I2C1_SDA、I2C1_SCL 并进行上拉
        SIM_PINSEL1 &= ～SIM_PINSEL1_I2C1PS_MASK;
        PORT_PUE1 |= 0x3;
        #endif
        #if(IIC_1_GROUP==2)
        //PTH3、PTH4 设置为 I2C1_SDA、I2C1_SCL
        SIM_PINSEL1 |= SIM_PINSEL1_I2C1PS_MASK;
        PORT_PUE1 |= 0x18000000;
         #endif
        I2C1_A1=address;                         //本机作为从机时的默认地址
        for(index=0;index<10;index+=2)
        {
            if(data[index]==BaudRate)
            {
               index++;
               I2C1_F=data[index];               //将定义的参数赋给 I2C1_F
               break;
            }
        }
        if(1 == Mode)                            //主机模式
        {
            BSET(I2C_C1_IICEN_SHIFT,I2C1_C1);    //开 i2c0 模块使能
            BSET(I2C_C1_TXAK_SHIFT,I2C1_C1);     //置位 i2c0 TXAK
        }
        else                                     //从机模式
        {
            BSET(I2C_C1_IICEN_SHIFT,I2C1_C1);    //使能 I2C 模块
            BSET(I2C_C1_IICIE_SHIFT,I2C1_C1);    //来 I2C 中断
            //BSET(I2C_C1_MST_SHIFT,I2C1_C1);    //设置成主机模式
            //BCLR(I2C_C1_TX_SHIFT,I2C1_C1);     //TX = 0,MCU 设置为接收模式
            //i=I2C0_D;                          //读出 IIC1D,准备接收数据
            //BCLR(I2C_C1_MST_SHIFT,I2C1_C1);  //MST 位由 1 变成 0，设置为从机模式
        }
```

```
    }
}

//=========================================================================
//函数名称：i2c_read1
//功能概要：从从机读 1 个字节数据
//参数说明：I2C_No:模块号，其取值为 0，1
//         DeviceAddr:设备地址范围为 1～255
//         AccessAddr:访问地址范围为 0～255
//         Data:带回收到的一个字节数据
//函数返回：为 0,成功读一个字节;为 1,读一个字节失败
//函数说明：内部调用 send_signal,wait
//=========================================================================
uint_8 i2c_read1(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 AccessAddr, uint_8 *Data)
{
    //获取 I2C 模块基址
    I2C_MemMapPtr num = i2c_get_base_address(I2C_No);
    BSET(I2C_C1_TX_SHIFT,I2C_C1_REG(num));     //TX = 1,MCU 设置为发送模式
    send_signal('S',I2C_No);                   //发送开始信号
    I2C_D_REG(num)  = DeviceAddr & 0xfe;       //发送设备地址,并通知从机接收数据
    if (wait('T',I2C_No))                      //等待一个字节数据传送完成
    {
        return 1;                              //没有传送成功,读一个字节失败
    }
    if (wait('A',I2C_No))                      //等待从机应答信号
    {
        return 1;                              //没有等到应答信号,读一个字节失败
    }
    I2C_D_REG(num)  = AccessAddr;              //发送访问地址
    if (wait('T',I2C_No))                      //等待一个字节数据传送完成
    {
        return 1;                              //没有传送成功,读一个字节失败
    }
    if (wait('A',I2C_No))                      //等待从机应答信号
    {
        return 1;                              //没有等到应答信号,读一个字节失败
    }
    //当 MCU 在主机模式下，向该位写 1 将产生一个重新开始信号
    BSET(I2C_C1_RSTA_SHIFT,I2C_C1_REG(num));
    I2C_D_REG(num) = DeviceAddr | 0x01;        //通知从机改为发送数据
    if (wait('T',I2C_No))                      //等待一个字节数据传送完成
    {
        return 1;                              //没有传送成功,读一个字节失败
    }
    if (wait('A',I2C_No))                      //等待从机应答信号
    {
        return 1;                              //没有等到应答信号,读一个字节失败
```

```
    }
    BCLR(I2C_C1_RSTA_SHIFT,I2C_C1_REG(num));  //TX = 0,MCU 设置为接收模式
    *Data = I2C_D_REG(num);                    //读出 IIC1D,准备接收数据
    if (wait('T',I2C_No))                      //等待一个字节数据传送完成
    {
        return 1;                              //没有传送成功,读一个字节失败
    }
    send_signal('O',I2C_No);                   //发送停止信号
    *Data = I2C_D_REG(num);                    //读出接收到的一个数据
     return 0;                                 //正确接收到一个字节数据
}

//=========================================================================
//函数名称：i2c_write1
//功能概要：向从机写 1 个字节数据
//参数说明：I2C_No:模块号，其取值为 0，1
//         DeviceAddr:设备地址范围为 1～255
//         AccessAddr:访问地址范围为 0～255
//         Data:要发给从机的 1 个字节数据
//函数返回：为 0,成功写一个字节;为 1,写一个字节失败
//函数说明：内部调用 send_signal,wait
//=========================================================================
uint_8 i2c_write1(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 AccessAddr, uint_8 Data)
{
    //获取 I2C 模块基址
    I2C_MemMapPtr num = i2c_get_base_address(I2C_No);
    BSET(I2C_C1_TX_SHIFT,I2C_C1_REG(num));//TX = 1,MCU 设置为发送模式
    send_signal('S',I2C_No);               //发送开始信号
    I2C_D_REG(num) = DeviceAddr & 0xfe;    //发送设备地址,并通知从机接收数据
    if (wait('T',I2C_No))                  //等待一个字节数据传送完成
        return 1;                          //没有传送成功,写一个字节失败
    if (wait('A',I2C_No))                  //等待从机应答信号
        return 1;                          //没有等到应答信号,写一个字节失败
    I2C_D_REG(num) = AccessAddr;           //发送访问地址
    if (wait('T',I2C_No))                  //等待一个字节数据传送完成
        return 1;                          //没有传送成功,写一个字节失败
    if (wait('A',I2C_No))                  //等待从机应答信号
        return 1;                          //没有等到应答信号,写一个字节失败
    I2C_D_REG(num) = Data;                 //写数据
    if (wait('T',I2C_No))                  //等待一个字节数据传送完成
        return 1;                          //没有传送成功,写一个字节失败
    if (wait('A',I2C_No))                  //等待从机应答信号
        return 1;                          //没有等到应答信号,写一个字节失败
    send_signal('O',I2C_No);               //发送停止信号
        return 0;
}
//=========================================================================
```

```
//函数名称：i2c_readn
//功能概要：从从机读 N 个字节数据
//参数说明：I2C_No:模块号，其取值为 0，1
//         DeviceAddr:设备地址范围为 1～255
//         AccessAddr:访问地址范围为 0～255
//         Data:读出数据的缓冲区
//         N:从从机读的字节个数
//函数返回：为 0,成功读 N 个字节;为 1,读 N 个字节失败
//函数说明：内部调用 i2c_read1
//=========================================================================
uint_8 i2c_readn(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 AccessAddr, uint_8
                 Data[], uint_8 N)
{
    uint_16 i, j;
    for (i = 0;i < N;i++)
    {
        for(j = 0;j <15*40 ;j++);        //最小延时(发送的每个字节之间要有时间间隔)
        if (i2c_read1(I2C_No,DeviceAddr, AccessAddr + i, &Data[i]))
            return 1;                    //其中一个字节没有接收到,返回失败标志:1
    }
    if (i >= N)
        return 0;                        //成功接收 N 个数据,返回成功标志:0
}


//=========================================================================
//函数名称：i2c_writen
//功能概要：向从机写 N 个字节数据
//参数说明：I2C_No:模块号，其取值为 0，1
//         DeviceAddr:设备地址范围为 1～255
//         AccessAddr:访问地址范围为 0～255
//         Data:要写入的数据的首地址
//         N:从从机读的字节个数
//函数返回：为 0,成功写 N 个字节;为 1,写 N 个字节失败
//函数说明：内部调用 i2c_write1
//=========================================================================
uint_8 i2c_writen(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 AccessAddr,uint_8
                  Data[], uint_8 N)
{
    uint_16 i, j;
    for (i = 0;i < N;i++)
    {
        for(j = 0;j < 15*40;j++);    //最小延时(发送的每个字节之间要有时间间隔)
        if (i2c_write1(I2C_No,DeviceAddr, AccessAddr + i, Data[i]))
            return 1;                //其中一个字节没有发送出去,返回失败标志:1
    }
    if (i >= N)
```

```
        return 0;                       //成功发送 N 个数据,返回成功标志:0
}
//=========================================================================
//函数名称：i2c_re_enable_int
//功能说明：打开 I2C 的 IRQ 中断
//函数参数：i2cNO:i2c 模块号，其取值为 0，1
//函数返回：无
//=========================================================================
void i2c_enable_re_int(uint_8 I2C_No)
{
    NVIC_EnableIRQ(I2C_No+8);
}

//=========================================================================
//函数名称：i2c_re_disable_int
//功能说明：关闭 I2C 的 IRQ 中断
//函数参数：i2cNO:i2c 模块号，其取值为 0,1
//函数返回：无
//=========================================================================
void i2c_disable_re_int(uint_8 I2C_No)
{
    NVIC_DisableIRQ(I2C_No+8);
}

//=========================================================================
//函数名称：i2c_get_base_address
//功能概要：获取 I2C 模块的基址
//参数说明：i2cNO:I2C 模块号，其取值为 0，1
//函数返回：I2C 模块的基址值
//=========================================================================
I2C_MemMapPtr i2c_get_base_address(uint_8 I2C_No)
{
       switch(I2C_No)
       {
          case 0:
                 return I2C0_BASE_PTR;
                   break;
          case 1:
                 return I2C1_BASE_PTR;
                    break;
    }
}
/=========================================================================
//函数名称：send_signal
//功能概要：根据需要产生开始或停止信号
//参数说明：I2C_No:模块号，其取值为 0，1
//         Signal: 'S'(Start),产生开始信号；'O'(Over),产生停止信号
```

```
//函数返回：无
//============================================================================
void send_signal(uint_8 Signal,uint_8 I2C_No)
{
    //获取 I2C 模块基址
    I2C_MemMapPtr num = i2c_get_base_address(I2C_No);
    if(num==I2C0)
    {
        if (Signal  == 'S')
        {
             //i2c0_c 主机模式选择位 MST 由 0 变为 1,可以产生开始信号
             BSET(I2C_C1_MST_SHIFT,I2C0_C1);
        }
        else if (Signal == 'O')
        {
             //主机模式选择位 MST 由 1 变为 0,可以产生停止信号
            BCLR(I2C_C1_MST_SHIFT,I2C0_C1);
        }
    }
    else if(num==I2C1)
    {
         if (Signal == 'S')
         {
             //i2c0_c 主机模式选择位 MST 由 0 变为 1,可以产生开始信号
            BSET(I2C_C1_MST_SHIFT,I2C1_C1);
          }
          else if (Signal == 'O')
         {
             //主机模式选择位 MST 由 1 变为 0,可以产生停止信号
             BCLR(I2C_C1_MST_SHIFT,I2C1_C1);
          }
    }
}
//============================================================================
//函数名称：  wait
//功能概要：在时限内,循环检测接收应答标志位,或传送完成标志位,判断 MCU
//          是否接收到应答信号或一个字节是否已在总线上传送完毕
//参数说明：I2C_No:模块号，其取值为 0，1
//          x:x = 'A'(Ack),等待应答;x = 'T'(Transmission),等待一个字节数据传输完成
//函数返回：0:收到应答信号或一个字节传送完毕;
//          1:未收到应答信号或一个字节没传送完
//============================================================================
uint_8 wait(uint_8 x,uint_8 I2C_No)
{
    uint_16 ErrTime, i;
    //获取 I2C 模块基地址
    I2C_MemMapPtr num = i2c_get_base_address(I2C_No);
```

```
    ErrTime = 255*10;                    //定义查询超时时限
    for (i = 0;i < ErrTime;i++)
    {
        if (x == 'A')                    //等待应答信号
        {
            if(( I2C_S_REG(num)  & I2C_S_RXAK_MASK)==0)
                return 0;                //传送完一个字节后,收到了从机的应答信号
         }
        else if (x == 'T')               //等待传送完成一个字节信号
        {
           if ((I2C_S_REG(num) & I2C_S_IICIF_MASK) != 0)
           {
                (I2C_S_REG(num) |=(0 | I2C_S_IICIF_MASK));  //清 IICIF 标志位
                    return 0;            //成功发送完一个字节
            }
         }
     }
    if (i >= ErrTime)
        return 1;                        //超时,没有收到应答信号或发送完一个字节
}

//=============================================================================
```

第 12 章

KEA128 的 MSCAN 总线开发方法

本章导读：本章阐述 KEA128 的 MSCAN 模块的构件化编程。主要内容有：12.1 节介绍 CAN 总线通用知识；12.2 节在简要阐述 MSCAN 编程要素的基础上，给出 KEA128 芯片 MSCAN 引脚复用及技术要点、MSCAN 构件接口函数说明及使用方法，这样读者就可以对 KEA128 芯片 MSCAN 进行编程操作；12.3 节给出 KEA128 芯片 MSCAN 模块驱动构件的制作方法，包括 MSCAN 寄存器的简介及封装要点；12.4 节给出使用 MSCAN 驱动构件设计。

本章参考资料：本章 12.2.1 节（时钟源选择、位速率及引脚复用）总结自《KEA128 参考手册》的第 32 章；12.3 节（寄存器简介）总结自《KEA128 参考手册》的第 32 章。

12.1 CAN 总线通用知识

控制器局域网（Controller Area Network，CAN）最早出现于 20 世纪 80 年代末，是德国 Bosch 公司为简化汽车电子中信号传输方式并减少日益增加的信号线而提出的。CAN 总线是一个单一的网络总线，所有的外围器件可以挂接在该总线上。

12.1.1 CAN 硬件系统的原理性电路

最简明的 CAN 硬件连接方法如图 12-1，把所有的 CAN_{TX} 线路经过快速二极管（如 1N4148 等）连接至数据线（以免输出引脚短路），CAN_{RX} 输入直接连接到这条数据线，数据线由一个上拉电阻拉至+5 V，以产生所需要的“1”电平。注意该电路中各节点的地是接在一起的，这个电路最大线长限制在 1 m 左右，主要用于在电磁干扰较弱环境下的近距离通信。

进行 CAN 通信节点调试时，可以利用图 12-1 这个简单且易于实现的电路。另外，可以利用该电路理解 CAN 总线的通信机制。该电路主要确保硬件连接能够通信无误的情况下，用于软件的调试和测试。但在实际应用中，一般需要采用差分硬件电路进行 CAN 通信，但软件与图 12-1 一致。

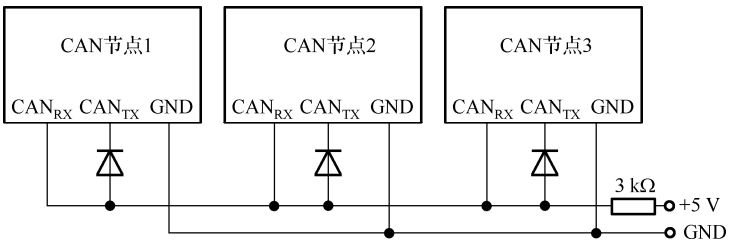


图 12-1　无需 CAN 收发器芯片的电路连接

12.1.2 CAN 总线的有关基本概念

1．报文、位速率

报文（Message）：是指在总线上传输的固定格式的信息，其长度是有限制的。当总线空闲时，总线上任何节点都可以发送新报文。报文被封装成帧（Frame）的形式在总线上传送。

位速率（Bit Rate）：是指 CAN 总线的传输速率。在给定的 CAN 系统中，位速率是固定、唯一的。CAN 总线上任意两个节点之间的最大传输距离与位速率有关，表 12-1 列出了距离与位速率的对应关系。这里的最大距离是指在不使用中继器的情况下两个节点之间的距离。

表 12-1 CAN 总线上任意两节点最大距离及位速率对应表

位速率/kbps	1000	500	250	125	100	50	20	10	5
最大距离/m	40	130	270	530	620	1300	3300	6700	10000

2．标识符 ID、优先权、仲裁

标识符 ID：CAN 节点的唯一标识。在实际应用时，应该给 CAN 总线上的每个节点按照一定规则分配一个唯一的 ID。每个节点发送数据时，发送的报文帧中含有发送节点的 ID 信息。

在 CAN 通信网络中，CAN 报文以广播方式在 CAN 网络上发送，所有节点都可以接收到报文，节点通过判断接收到的标识符 ID 决定否接收该报文。报文标识符 ID 的分配规则一般在 CAN 应用层协议实现（比较著名的 CAN 应用层协议为 CANopen 协议和 DeviceNet 等）。由于 ID 决定报文发送的优先权，因此 ID 的分配规则在实际应用中必须给予重视。一般可以用标识符的某几位代表发送节点的地址。接收到报文的节点可以通过解析接收报文的标识符 ID，来判断该报文来自哪个节点，属于何种类型的报文等。下面给出 CANopen 协议最小系统配置的一个 ID 分配方案，供实际应用时参考。

D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
功能代码				节点地址						

该分配方案是一个面向设备的标识符分配方案，该方案通过 4 位的功能代码区分 16 种不同类型的报文，有 7 位节点地址，可表达 128 个节点。但要注意到 CAN 协议中，要求 ID 的高 7 位不能同时为 1。报文标识符 ID 的分配方法应遵循以下原则：在同一系统中，必须保证节点地址唯一，这样每个报文的 ID 也就唯一了。

优先权（Priorities）：在总线访问期间，报文的标识符 ID 定义了一个静态的报文优先权。在 CAN 总线上发送的每一个报文都具有唯一的一个 11 位或 29 位的标识符 ID，在总线仲裁时，显性位（逻辑 0）的优先权高于隐性位（逻辑 1），从而标识符越小，该报文拥有越高的优先权，因此一个拥有全 0 标识符的报文具有总线上的最高级优先权。

当有两个节点同时进行发送时，必须通过“无损的逐位仲裁[①]”方法来使得有最高优先权的报文优先发送。

① “无损的逐位仲裁”：当总线上出现报文冲突时，仲裁机制逐位判断标识符，实现高优先权的报文能够不受任何损坏地优先发送。

仲裁（Arbitration）：总线空闲时，总线上任何节点都可以开始发送报文，若同时有两个或两个以上节点开始发送，总线访问冲突运用逐位仲裁规则，借助于标识符 ID 解决。仲裁期间，每一个发送器都对发送位电平与总线上检测到的电平进行比较，若相同，则该节点继续发送。当发送的是“1”，而监视到的是“0”时，则该节点失去仲裁，退出发送状态。举例说明，若某一时刻有两个 CAN 节点 A、B 同时向总线发送报文，A 发送报文的 ID 为 0b00010000000，B 发送报文的 ID 为 0b01110000000。由于节点 A、B 的 ID 的第 10 位都为“0”，而 CAN 总线是逻辑与的，因此总线状态为“0”，此时两个节点检测到总线位和它们发送的位相同，因此两个节点都认为是发送成功，都继续发送下一位。发送第 9 位时，A 发送一个“0”，而 B 发送一个“1”，此时总线状态为“0”。此时 A 检测到总线状态“0”与其发送位相同，因此 A 认为它发送成功，并开始发送下一位。但此时 B 检测到总线状态“0”与其发送位不同，它会退出发送状态并转为监听方式，且直到 A 发送完毕，总线再次空闲时，它才试图重发报文。

3．帧结构

CAN 总线协议中有四种报文帧（Message Frame），它们分别是数据帧、远程帧、错误帧、过载帧，其中数据帧和远程帧，与用户编程相关，错误帧和过载帧由 CAN 控制硬件处理，与用户编程无关。

在 CAN 节点之间的通信中，要将数据从一个节点发送器传输到另一个节点的接收器，必须发送数据帧。而总线上节点发送远程帧目的在于请求发送具有同一标识符的数据帧。

数据帧由 7 个不同的位场组成：帧起始（Start Of Frame symbol，SOF）、仲裁场、控制场、数据场、CRC 场、应答场、帧结束（End Of Frame，EOF），如图 12-2 所示。

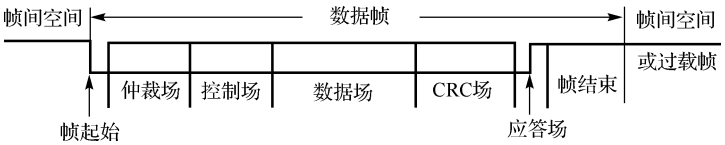


图 12-2　数据帧组成

根据仲裁场不同，在 CAN2.0B 中定义标准帧与扩展帧两种帧格式。标准帧的标识符 ID 为 11 位，扩展帧的标识符 ID 为 29 位（11 位标准 ID+18 位扩展 ID）。

12.2　MSCAN 驱动构件及使用方法

MSCAN 模块完全支持 CAN 协议 2.0A/B 版：标准和扩展数据帧、0～8 字节数据长度、高达 1 Mbps 的可编程波特率、支持远程帧；拥有 5 个具有 FIFO 存储机制的接收缓冲器和 3 个具有使用“本地优先”概念的内部优先顺序的发送缓冲器；三种可编程的掩码标识符过滤器，2 个全尺寸（32 位）扩展标识符过滤器、4 个 16 位过滤器和 8 个 8 位过滤器，过滤掉无效的帧数据；可编程选择采用总线时钟或振荡器时钟作为 MSCAN 时钟源；内置定时器模块，用于设置接收和发送的时间戳；共 3 种低功耗模式，睡眠、断电和 MSCAN 使能；可以全局初始化和配置寄存器。

12.2.1 KEA128 芯片 MSCAN 编程知识要素

1．KEA128 芯片 MSCAN 模块的时钟源选择

MSCAN 模块有两种时钟源选择，分别是总线时钟（Bus Clock）和振荡器时钟（Oscillator Clock，OSC）。MSCAN 控制寄存器 1（MSCAN_CANCTL1）的时钟源位（CLKSRC）可控制选择时钟源。如果总线时钟是经过锁相环（PhaseLockedLoop，PLL）产生的，CAN 总线传输速率很高时会存在抖动问题，建议选择 OSC 时钟源。若使用 OSC 时钟需要外接晶振，而且 KEA128 芯片的总线时钟是经过锁频环（Frequency Lock Loop，FLL）产生的，所以在样例程序中 MSCAN 模块的时钟源为总线时钟。

2．KEA128 芯片 MSCAN 模块的位速率

位速率（Bit Rate）也称为波特率，是指每秒发送的位数。位时间（Bit Time），是指发送一位所需要的时间。实际工作过程的位时间与系统设定的位时间稍有偏差，把理想情况下相应的位速率称为标称位速率，位时间称为标称位时间（Nominal Bit Time）。

$$\text{位速率}=1/\text{位时间} \tag{12-1}$$

时间冲量 T_q（Time Quanta）是 MSCAN 处理需要的原子单位时间，从 CANCLK 的预分频得到时间冲量。预分频值由 MSCAN 总线时序寄存器 0（MSCAN_CANBTR0）的 BRP 决定，取值范围为 1～64。

$$T_q=\text{CANCLK}/(\text{预分频器值}) \tag{12-2}$$

位时间再分成三段，如 Bosch CAN 规范所述。

（1）SYNC_SEG：该段为 1 个固定长度的 T_q，信号边沿预计出现在本段。

（2）时段 1（TSEG1）：本段包括 CAN 标准的 PROP_SEG（传播段）和 PHASE_SEG1（相位段 1），可包含 4～16 个 T_q。

（3）时段 2（TSEG1）：本段包括 CAN 标准的 PHASE_SEG2（相位段 2），可包含 2～8 个 T_q。

MSCAN 总线时序寄存器 1（MSCAN_CANBTR1）的 TSEG1 和 TSEG2 字段分别设置时段 1 和时段 2 包含的 T_q 的个数。假设时段 1 和时段 2 包含的 T_q 的个数分别为 T_1 和 T_2。

$$\text{位时间}=\text{SYNC_SEG}+\text{TSEG1}+\text{TSEG2}=(1+T_1+T_2)\,T_q \tag{12-3}$$

由式（12-1）、式（12-2）和式（12-3）可得，

$$\text{位速率}=T_q/(1+T_1+T_2)$$

网上光盘的样例程序中设置 MSCAN 的 CANCLK 为 24 MHz、预分频器值为 2、时间段 1 为 16 个 T_q，时间段 2 为 7 个 T_q，因此位速率为 500 kbps。

3．KEA128 芯片 MSCAN 模块的标识符 ID 匹配

MSCAN 模块使用标识符接收过滤器进行标识符匹配处理，这种模式可以降低 CPU 的负载。MSCAN 模块的标识符接收过滤模式有四种，分别为 2 个 32 位的接收过滤器、4 个 16 位的接收过滤器、8 个 8 位接收过滤器和过滤器关闭模式。在 MSCAN 标识符接收控制寄存

器（MSCAN_CANIDAC）的标识符接收模式（IDAM）控制选择模式。在过滤器关闭模式下，接收不到消息。

下面以 32 位的接收过滤器标准帧为例，介绍 ID 过滤的方法，如图 12-3 所示。接收标准标识符寄存器 RSIDR0-RSIDR1 用于存放 MSCAN 接收到的消息中的标准标识符 ID，用户不需要关注；标识符接收寄存器（CANIDARn）存放用户预设的想要接收的节点的 ID；标识符掩码寄存器（CANIDMRn）用于设置是否将 RSIDR 与 CANIDAR 寄存器中的对应位进行匹配处理。

图 12-3 为第一个 32 位的过滤器段（CANIDAR0～3，CANIDMR0～3）产生过滤器 0 命中的示意图，而第二个 32 位的过滤器段（CANIDAR4～7，CANIDMR4～7）产生过滤器 1 命中。标准标识符 ID 有 11 位，RSIDR1 中存放标准 ID 的第 0～2 位，RSIDR0 中存放标准 ID 的第 3～10 位。

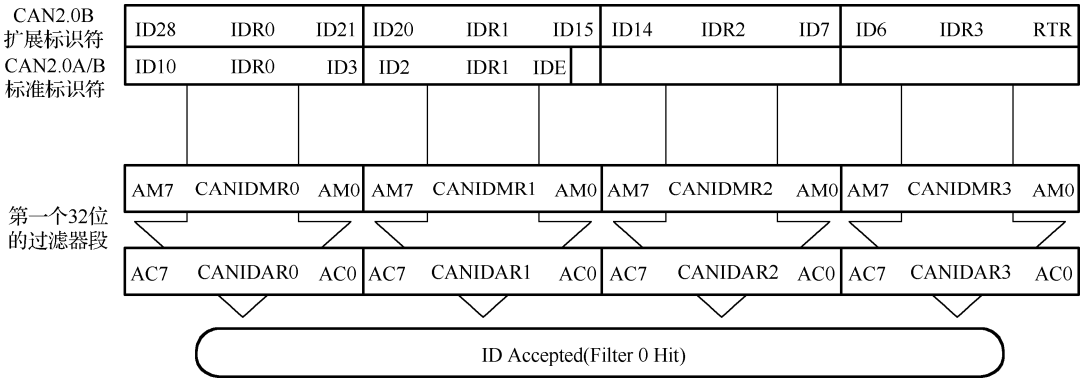


图 12-3　采用 32 位掩码的标识符接收过滤器

以如下代码为例，采用一个 32 位过滤器进行标准标识符过滤，第一个过滤器段的 CANIDAR0 寄存器（CANIDAR_BANK_1[0]）用于过滤标准帧 ID 的 3～10 位，CANIDAR1 寄存器（CANIDAR_BANK_1[1]）用于过滤标准帧 ID 的 0～2 位。因此，将用户预设的想要接收的节点的 ID（acc_id）进行移位处理分别放在 CANIDAR_BANK_1[0] 和 CANIDAR_BANK_1[1]中，并设置 CANIDMR_BANK_1[0]和 CANIDMR_BANK_1[1]进行匹配。若二者的值完全匹配则表示对应数据可以被接收，此时即表示过滤器命中。注意：MSCAN 有 8 个命中过滤器 0～7，在 MSCAN 标识符接收控制寄存器（MSCAN_CANIDAC）的 IDHIT 位查看哪个过滤器命中。

```
//设置标识符接收寄存器的值
p->CANIDAR_BANK_1[0]=acc_id>>3;
p->CANIDAR_BANK_1[1]=acc_id<<5;
p->CANIDAR_BANK_1[2]=0xFF;
p->CANIDAR_BANK_1[3]=0xFF;
//设置掩码寄存器的值
p->CANIDMR_BANK_1[0]=0x00;   //0：对应位开启匹配，1：对应位关闭匹配
p->CANIDMR_BANK_1[1]=0x1F;
p->CANIDMR_BANK_1[2]=0xFF;
p->CANIDMR_BANK_1[3]=0xFF;
```

4．KEA128 芯片 MSCAN 引脚

MSCAN 模块的发送数据引脚为 CAN_TX，接收数据引脚为 CAN_RX。它们并不是固定在哪几个引脚上，而是通过系统集成模块（SIM）提供的引脚选择寄存器 1（Pin Selection Register1，SIM_PINSEL1）中的 MSCANPS 位编程来设定的。当 MSCANPS=0 时，CAN_TX 映射到引脚 PTC7 上，CAN_RX 映射到引脚 PTC6 上；当 MSCANPS=1 时，CAN_TX 映射到引脚 PTE7 上，CAN_RX 映射到引脚 PTH2 上。KEA128 芯片默认使用 PTC7 和 PTC6 分别作为 CAN_TX 和 CAN_RX 引脚。

12.2.2 MSCAN 构件头文件及使用方法

在了解 KEA128 芯片 MSCAN 基本特性的基础上，本节先行给出 MSCAN 构件封装的头文件，目标是读者看完本小节就能够编程，把 MSCAN 构件使用起来。

1．MSCAN 驱动构件头文件（can.h）

```
//=============================================================================
//文件名称：can.h
//功能概要：CAN 底层驱动构件头文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-5-11  V1.5
//芯片类型：KEA128
//=============================================================================

#ifndef CAN_H                //防止重复定义（can_H  开头)
#define CAN_H

#include "includes.h"        //包含头文件

#define LOOP_MODE     0
#define NORMAL_MODE   1

//定义 CAN 通信的帧结构
typedef struct CanMsg
{
    uint_32 m_ID;            //msg 发送方 ID
    uint_8 m_IDE;            //扩展帧为 1，标准帧为 0
    uint_8 m_RTR;            //远程帧为 1，数据帧为 0
    uint_8 m_data[8];        //帧数据
    uint_8 m_dataLen;        //帧数据长度
    uint_8 m_priority;       //发送优先级
} CANMsg;

//=============================================================================
//函数名称：CAN_Init
//功能概要：初始化 MSCAN 模块，采用 24 MHz 总线时钟作为模块时钟，波特率固定为 500 kbps
//参数说明：mode:工作模式选择:
```

```
//             LOOP_MODE(0)   - 回环模式
//             NORMAL_MODE(1) - 普通模式
//         open_filter:接收过滤器开关
//             1 - 开启过滤器，只接收标识符 ID 为 acc_id 的帧
//             0 - 关闭过滤器，接收所有帧
//         acc_id:接受 ID，接收过滤器只接收标识符符合此 ID 的帧，过滤其他 ID 的帧
//函数返回：无
//============================================================================
void CAN_Init(uint_8 mode,uint_8 open_filter,uint_32 acc_id);

//============================================================================
//函数名称：CAN_Fill_STD_Msg
//功能概要：填充一个 CAN 数据包
//参数说明：message:待发送 CAN 数据包结构体的首地址
//          id:待发送 CAN 标准帧的标识符 ID（当前节点 ID）
//          data:待发送数据缓冲区首地址
//          len:待发送数据长度（<=8 Bytes）
//函数返回：0 - 填充成功
//          1 - 数据长度输入错误
//============================================================================
uint_8 CAN_Fill_STD_Msg(CANMsg *message,uint_32 id,uint_8 *data,uint_8 len);

//============================================================================
//函数名称：CAN_Send_Msg
//功能概要：CAN 发送数据包函数
//参数说明：sendmsg:所发送的 CAN 数据包的首地址
//函数返回：0 - 发送成功
//          1 - 数据帧长度错误
//          2 - 发送帧为扩展帧
//============================================================================
uint_8 CAN_Send_Msg(CANMsg *sendmsg);

//============================================================================
//函数名称：CAN_Receive_Msg
//功能概要：接收 CAN 数据包函数
//参数说明：receiveFrame:存放待接收 CAN 数据包缓冲区的首地址
//函数返回：0 - 成功接收标准帧
//          1 - 未接收到帧
//          2 - 收到扩展帧
//============================================================================
uint_8 CAN_Receive_Msg(CANMsg *recvmsg);

#endif     //防止重复定义（CAN_H  结尾)
```

2．MSCAN 驱动构件的使用方法

下面以两个 CAN 节点进行通信为例，简要介绍 MSCAN 驱动构件的使用方法。两个 CAN

节点分别标记为节点 A 和节点 B。两个节点可使用 2 个带 MSCAN 通信接口的 KEA128 实验板来代替。

标识符的分配如下。

节点 A：ID=0x0A；节点 B：ID=0x0B。

假设两个节点通信均传输标准数据帧，两个节点共同向 CAN 总线发送数据，各节点通过过滤机制接收对应的数据包：节点 A 将接收节点 B 的数据包，节点 B 将接收节点 A 的数据包。两个节点均通过中断方式接收来自 CAN 总线上的数据，接收到对应的数据包后，将数据通过串口发送到 PC。上述通信过程的实现步骤如下。

1）节点 A 设置

（1）设置节点 A 的 ID 为 0x0A，接收 ID 为 0x0B，设置语句如下。

```
//定义宏常数
#define ACC_ID    0x0B              //接收 ID
#define LOCAL_ID  0x0A              //本机 ID
……
//正常模式
#define LOOP_MODE     0
#define NORMAL_MODE  1

CAN_Init(NORMAL_MODE,1,ACC_ID);     //CAN 模块初始化为正常模式，开启 32 位过滤器
```

（2）节点 A 向节点 B 发送数据包为“Node_A”，设置语句如下。

```
……
uint_8 sendData[]="Node_A";         //发送的数据内容
uint_8 sendDataLength;              //CAN 发送数据长度
CANMsg sendMsg;                     //待发送数据包
//填充 CAN 发送数据包
(void)CAN_Fill_STD_Msg(&sendMsg, LOCAL_ID, sendData, sendDataLength);
//发送数据包
SNDFlag = CAN_Send_Msg(&sendMsg);
……
```

（3）在接收中断服务例程中，节点 A 接收节点 B 发送的数据包“Node_B”，设置语句如下。

```
//接收 CAN 帧
RCVFlag = CAN_Receive_Msg(&recvMsg);
```

2）节点 B 设置

（1）设置节点 B 的 ID 为 0x0B，接收 ID 为 0x0A，设置语句如下。

```
//定义全局变量
#define ACC_ID    0x0A              //接收 ID
#define LOCAL_ID  0x0B              //本机 ID
……
//正常模式
#define LOOP_MODE     0
```

```
#define NORMAL_MODE  1

CAN_Init(NORMAL_MODE,1,ACC_ID);  //CAN 模块初始化为正常模式，开启 32 位过滤器
```

（2）节点 B 向节点 A 发送数据包为“Node_B”，设置语句如下。

```
……
uint_8 sendData[] = "Node_B";          //发送的数据内容
uint_8 sendDataLength;                 //CAN 发送数据长度
CANMsg sendMsg;                        //待发送数据包
//填充 CAN 发送数据包
(void)CAN_Fill_STD_Msg(&sendMsg, LOCAL_ID, sendData, sendDataLength);
//发送数据包
SNDFlag = CAN_Send_Msg(&sendMsg);
……
```

（3）在接收中断服务例程中，节点 B 接收节点 A 发送的数据包“Node_A”，设置语句如下。

```
//接收 CAN 帧
RCVFlag = CAN_Receive_Msg(&recvMsg);
```

12.3　MSCAN 驱动构件制作方法

12.3.1　MSCAN 寄存器简介

MSCAN 模块的基址取决于 MCU 部分内存映像，从模块基址开始一共有 64 字节的地址空间用于 MSCAN 模块的寄存器定义，模块的基地址为 0x4002_4000，所有寄存器均为 8 位寄存器。将寄存器按功能整理如表 12-2 所示。

表 12-2　MSCAN 寄存器简介

寄 存 器 名	功 能 描 述
MSCAN 控制寄存器 *n*（MSCAN_CANCTLn）（*n*=0,1）	控制 MSCAN 模块的基本功能
MSCAN 总线时序寄存器 *n*（MSCAN_CANBTRn）（*n*=0,1）	配置与 MSCAN 模块相关的时钟周期
接收扩展标识符寄存器 *n*（MSCAN_REIDRn）（*n*=0～3）	扩展帧标识符
接收标准标识符寄存器 *n*（MSCAN_RSIDRn）（*n*=0,1）	标准帧标识符
MSCAN 标识符接收控制寄存器（MSCAN_CANIDAC）	控制过滤器接收模式，读取命中的过滤器
MSCAN 第一块（First Bank）标识符接收寄存器 *n*（MSCAN_CANIDARn）	存放用户预设的想要接收的节点的 ID
MSCAN 第二块（Second Bank）标识符接收寄存器 *n*（MSCAN_CANIDARn）	
MSCAN 第一块（First Bank）标识符掩码寄存器 *n*（MSCAN_CANIDMRn）	用于设置是否将 RSIDR 与 CANIDAR 寄存器中的对应位进行匹配处理。
MSCAN 第二块(Second Bank)标识符掩码寄存器 *n*(MSCAN_CANIDMRn）	
接收扩展数据段寄存器 *n*（MSCAN_REDSRn）	存放接收的数据
接收数据长度寄存器（MSCAN_RDLR）	接收的数据长度，不能超过 8
MSCAN 接收器标志寄存器（MSCAN_CANRFLG）	包含接收数据标志位

续表

寄 存 器 名	功 能 描 述
MSCAN 接收器中断使能寄存器 （MSCAN_CANRIER）	包含接收中断使能位
发送扩展标识符寄存器 *n*（MSCAN_TEIDRn）（n=0～3）	扩展帧标识符
发送标准标识符寄存器 *n*（MSCAN_TSIDRn）（n=0,1）	标准帧标识符
MSCAN 发送缓冲区选择寄存器（MSCAN_CANTBSEL）	选择发送缓冲区
发送缓冲区优先级寄存器（MSCAN_TBPR）	设置发送缓冲区的优先级
发送扩展数据段寄存器 *n*（MSCAN_TEDSRn）	存放接收的数据
发送数据长度寄存器（MSCAN_TDLR）	发送的数据长度，不能超过 8
MSCAN 发送器标志寄存器（MSCAN_CANTFLG）	包含发送数据标志位
MSCAN 发送器中断使能寄存器（MSCAN_CANTIER）	包含发送中断使能位
MSCAN 发送消息终止请求寄存器 （MSCAN_CANTARQ）	包含终止请求位
MSCAN 发送消息中止应答寄存器（MSCAN_CANTAAK）	包含终止应答位
接收时间戳寄存器 H（MSCAN_RTSRH）	通过 MSCAN 内部定时器获得接收数据的时间戳
接收时间戳寄存器 L（MSCAN_RTSRL）	
发送时间戳寄存器高（MSCAN_TTSRH）	通过 MSCAN 内部定时器获得发送数据的时间戳
发送时间戳寄存器低（MSCAN_TTSRL）	
MSCAN 接收错误计数器（MSCAN_CANRXERR）	反映 MSCAN 接收错误计数器的状态
MSCAN 发送错误计数器（MSCAN_CANTXERR）	反映 MSCAN 发送错误计数器的状态
MSCAN 杂项寄存器（MSCAN_CANMISC）	配置 MSCAN 模块是否保持处于 bus-off 状态

控制寄存器的设置是在初始化阶段完成的，此时 MSCAN 模块与 CAN 总线尚未同步，而配置寄存器的设置可以在同步之后进行改变。要改变配置寄存器，首先要通过置位 MSCAN_CANCTL0 的 INITRQ 位使 MSCAN 进入初始化模式。

由于该模块寄存器内容较多，本节仅介绍 MSCAN 模块中的部分寄存器和寄存器位，MSCAN 模块其它寄存器的详细内容请参阅网上“SD-FSL-KEA-CD(V1.1)-20150623”提供的资料。

1．MSCAN 控制寄存器 0（CANCTL0）

MSCAN 控制寄存器 0 提供 MSCAN 模块的各种控制，偏移地址为 0x00。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	RXFRM	RXACT	CSWAI	SYNCH	TIME	WUPE	SLPRQ	INITRQ
写		—		—				
复位	0							1

D7（RXFRM）—接收帧标志位。该位只读而且只能清 0，当正确接收到一个有效报文时该位置 1，而不依赖过滤器的设置。被置 1 后需要软件设置或复位才能清 0。对该位写“1”可以清 0，写“0”无效。该位在自环模式（Loop Back）下无效。1=接收到有效报文；0=未接收到有效报文。

D6（RXACT）—接收激活状态位。该只读位指示 MSCAN 正在接收数据，该位由接收器控制，在自回环模式（Loop Back）下无效。RXACT=1，MSCAN 正在接收数据（或者仲裁丢失的时候）；RXACT=0，MSCAN 正在发送数据或者空闲。

D5（CSWAI）－在等待模式下 CAN 停止位。该位置 1，则在低功耗的等待模式下关闭

所有 MSCAN 模块总线接口的时钟。CSWAI=1，在等待模式下模块停止时钟；CSWAI=0，在等待模式下模块不受影响。

D4（SYNCH）—同步状态位。该只读位指示 MSCAN 模块是否同步于 CAN 总线。该位由 MSCAN 设置和清 0。SYNCH=1，MSCAN 模块同步于 CAN 总线；SYNCH=0，MSCAN 模块与 CAN 总线不同步。

D3（TIME）—定时器使能位。该位激活一个内部 16 位自由运行定时器模块。定时器模块激活后在每个发送/接收缓冲区内的消息上加上 16 位的时间戳。一旦消息被 CAN 总线接收，时间戳将写在相应缓冲区的最高字节（0x000E，0x000F）。TIME=1，使能 MSCAN 内部定时器模块；TIME=0，禁止 MSCAN 内部定时器模块。

D2（WUPE）—唤醒使能位。该位允许 MSCAN 模块在检测到 CAN 总线通信时从睡眠模式重启。WUPE=1，允许 MSCAN 唤醒；WUPE=0，不允许 MSCAN 唤醒，使 MSCAN 忽略 CAN 总线通信。

D1（SLPRQ）—睡眠模式请求位。该位要求 MSCAN 模块进入睡眠模式。SLPRQ=1，提出睡眠模式请求，当 CAN 总线空闲时 MSCAN 模块进入睡眠模式；SLPRQ=0，MSCAN 模块正常工作。

D0（INITRQ）—初始化模式请求位。INITRQ=1，提出 MSCAN 模块初始化模式请求；INITRQ=0，MSCAN 模块正常工作。

2．MSCAN 控制寄存器 1（CANCTL1）

MSCAN 控制寄存器 1 提供 MSCAN 模块的各种控制和握手状态信息，偏移地址为 0x01。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	CANE	CLKSRC	LOOPB	LISTEN	BORM	WUPM	SLPAK	INITAK
写							—	—
复位	0			1	0			1

D7（CANE）—MSCAN 模块使能位。CANE=1，MSCAN 模块使能；CANE=0，MSCAN 模块禁止。

D6（CLKSRC）—MSCAN 时钟源位。CLKSRC=1，采用总线时钟；CLKSRC=0，采用振荡器时钟。

D5（LOOPB）－回环测试模式位。LOOPB=1，回环测试模式使能；LOOPB=0，回环测试模式禁止。

D4（LISTEN）—监听模式位。该位使得 MSCAN 模块作为 CAN 总线监听器。当该位置 1，所有 ID 匹配的有效 CAN 信息被接收，但是不发送应答或错误帧，错误计数被冻结。在监听模式，MSCAN 不能发送消息。LISTEN=1，监听模式激活；LISTEN=0，正常工作。

D3（BORM）—总线关闭恢复位。该位配置 MSCAN 的总线关闭恢复模式。BORM=1，用户请求时总线关闭恢复；BORM=0，自动总线关闭恢复（参见 Bosch CAN 2.0A/B）。

D2（WUPM）—唤醒模式位。该位定义低通滤波器是否应用于保护 MSCAN 模块防止被

假唤醒。WUPM=1，只有当 CAN 总线上产生一个长度为 T_{wup} 的显性脉冲时，唤醒 MSCAN；WUPM=0，CAN 总线上收到任何有效电平后，唤醒 MSCAN。

D1（SLPAK）—睡眠模式应答位。该位表明 MSCAN 模块是否进入睡眠模式，作为 SLPRQ 睡眠模式请求的应答握手标志。SLPAK=1，MSCAN 模块进入睡眠模式；SLPAK=0，MSCAN 模块正常工作。

D0（INITAK）—初始化模式应答位。该位指示 MSCAN 模块是否进入初始化模式，作为 INITRQ 初始化模式请求的应答握手标志。INITAK=1，MSCAN 模块已经进入初始化模式；INITAK=0，MSCAN 模块正常工作。

3．MSCAN 总线时序寄存器 0（CANBTR0）

MSCAN 总线时钟寄存器 0 配置 MSCAN 模块的各种总线时序参数，偏移地址为 0x02。

D7～D6（SJW）—同步补偿宽度位。定义了一个可以被缩短或延长来重新同步总线数据发送的时间冲量（T_q）时钟周期的最大数，取值范围为 00～11，分别表示 0～4 个 T_q 时钟周期。

D5～D0（BRP）—波特率预分频位。BRP 字段决定 T_q 时钟，取值范围为 00000～11111。

4．MSCAN 总线时序寄存器 1（CANBTR1）

MSCAN 总线时钟寄存器 1 配置 MSCAN 模块的各种总线时序参数，偏移地址为 0x03。

D7（SAMP）—采样位。该位决定位时间内 CAN 总线的采样次数。在很高的位速率下，建议设置 SAMP=0。SAMP=1，位时间内，3 次采样。这种情况下 PHASE_SEG1 至少为 2 个时间冲量；SAMP=0，位时间内，1 次采样。

D6～D4（TSEG2）—时间段 2。在位时间内，确定每个位时间的时钟周期数和采样点的位置，取值范围是 000～111，分别表示 0～8 个 T_q 时钟周期。

D3～D0（TSEG1）—时间段 1。在位时间内，确定每个位时间的时钟周期数和采样点的位置，取值范围是 0000～1111，分别表示 1～16 个 T_q 时钟周期。

位时间由晶振频率 f_{CANCLK}、波特率预分频和每位的 T_q 数量决定。

$$\text{位时间} = \text{预分频值} \times T_q\text{数} / f_{CANCLK}$$

5．MSCAN 接收器标志寄存器（CANRFLG）

寄存器中的标志只有在导致置位的条件不再有效时或者由软件来清 0，每个标志对应 CANRIER 寄存器的一个中断使能位，其偏移地址为 0x04。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	WUPIF	CSCIF	RSTAT		TSTAT		OVRIF	RXF
写			—		—			
复位	0							

D7（WUPIF）—唤醒中断标志位。WUPIF=1，MSCAN 检测到总线活动并请求唤醒；WUPIF=0，在睡眠模式下没有检测到唤醒请求。

D6（CSCIF）—CAN 状态变化中断标志位。CSCIF=1，MSCAN 改变当前 CAN 总线状态；CSCIF=0，从上次中断后 CAN 总线状态没有变化。

D5～D4（RSTAT）－接收器状态位。RSTAT=00，表示 RxOK，0≤接收错误计数＜96；RSTAT=01，表示 RxWRN，96≤接收错误计数＜128；RSTAT=10，表示 RxERR，128≤接收错误计数；RSTAT=11，Bus-Off，发送错误计数＞255。

D3～D2（TSTAT）—发送器状态位。TSTAT=00，表示 TxOK，0≤接收错误计数＜96；TSTAT=01，表示 TxWRN，96≤接收错误计数＜128；TSTAT=10，表示 TxERR，128≤接收错误计数＜256；TSTAT=11，Bus-Off，256≤发送错误计数。

D1（OVRIF）—溢出中断标志位。OVRIF=1，检测到数据溢出；OVRIF=0，无数据溢出。

D0（RXF）—接收缓冲区满标志位。RXF=1，接收器 FIFO 非空。RxFG 中有一个新报文可访问；RXF=0，RxFG 中无新报文。

6．MSCAN 接收器中断使能寄存器（CANRIER）

该寄存器包含在 CANRFLG 中断标志寄存器中的中断使能位，其偏移地址为 0x05。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	WUPIE	CSCIE	RSTATE		TSTATE		OVRIE	RXFIE
写								
复位	0							

D7（WUPIE）—唤醒中断使能位。WUPIE=1，一个唤醒事件将产生一个唤醒中断请求；WUPIE=0，一个唤醒事件不产生中断请求。

D6（CSCIE）—CAN 状态变化中断使能位。CSCIE=1，CAN 总线状态变化事件将产生一个错误中断请求；CSCIE=0，CAN 总线状态变化事件不产生中断请求。

D5～D4（RSTATE）—接收器状态变化使能位。RSTATE=11，接收器所有状态变化都产生 CSCIF 中断；RSTATE=10，仅当接收器进入或离开 RxERR 或 Bus-Off 状态将产生 CSCIF 中断；RSTATE=01，仅当接收器进入或离开 Bus-Off 状态将产生 CSCIF 中断；RSTATE=00，不产生因接收器状态变化引起的 CSCIF 中断。

D3～D2（TSTATE）—发送器状态变化使能位。TSTATE=11，发送器所有状态变化都产生 CSCIF 中断；TSTATE=10，仅当发送器进入或离开 TxERR 或 Bus-Off 状态都将产生 CSCIF 中断；TSTATE=01，仅当发送器进入或离开 Bus-Off 状态都产生 CSCIF 中断；TSTATE=00，不产生因发送器状态变化引起的 CSCIF 中断。

D1（OVRIF）—溢出中断使能位。OVRIF=1，数据溢出事件将产生一个错误中断请求；OVRIF=0，数据溢出事件不产生中断请求。

D0（RXFIE）—接收缓冲区满中断使能位。RXFIE=1，接收缓冲区满事件将产生一个接收器中断请求；RXFIE=0，接收缓冲区满事件不产生中断请求。

7．MSCAN 发送器标志寄存器（CANTFLG）

发送缓冲区空标志在 CANTIER 寄存器中都有一个相应的中断使能位，偏移地址为 0x06。

D2～D0（TXE）—发送缓冲区空标志位。TXEx=1（x=2、1、0）、相应发送消息缓冲区空；TXEx=0（x=2、1、0），相应发送消息缓冲区满。

8．MSCAN 发送器中断使能寄存器（CANTIER）

CANTIER 寄存器的偏移地址为 0x07。

D2～D0（TXEIE）—发送缓冲区空中断使能位。TXEIEx=1（*x*=2、1、0），相应的发送缓冲区空事件将产生一个发送中断请求；TXEIEx=0（*x*=2、1、0），相应的发送缓冲区空事件不产生中断请求。

9．MSCAN 发送器消息中止请求寄存器（CANTARQ）

CANTARQ 寄存器提供队列消息的发送中止请求，偏移地址为 0x08。

D2～D0（ABTRQ）—中止请求位。ABTRQx=1（*x*=2、1、0），相应的发送缓冲区提交中止请求；ABTRQx=0（*x*=2、1、0），相应的发送缓冲区无中止请求。

10．MSCAN 发送器消息中止应答寄存器（CANTAAK）

该标志位确认消息由于 CPU 挂起中止请求被中止。特定的消息缓冲区被标记为空后，应用程序使用该标志位，以确定消息是否已成功中止还是已发送。当相应的 TXE 标志被清 0，ABTAKx 标志位会清 0。偏移地址为 0x09。

D2～D0（ABTAK）—中止应答位。ABTAKx=1（*x*=2、1、0），相应的发送缓冲区被中止；ABTAKx=0（*x*=2、1、0），相应的发送缓冲区没有被中止。

11．MSCAN 发送缓冲区选择寄存器（CANTBSEL）

CANTBSEL 寄存器表明选中的发送消息缓冲区，偏移地址为 0x0A。

D2～D0（TX）—发送缓冲区选择标志位。TXx=1（*x*=2、1、0），相应的发送缓冲区被选中；TXx=0（*x*=2、1、0），相应的发送缓冲区没有被选中。

12．MSCAN 标识符接收控制寄存器（CANTIDAC）

CANTIDAC 寄存器用于控制标识符接收，偏移地址为 0x0B。

D5～D4（IDAM）—标识符接收模式位。CPU 设置该字段来定义标识符接收过滤器的模式。在过滤器关闭模式，不接收消息，使得前台缓冲区永远不会重新加载报文。IDAM=00，选择两个 32 位接收过滤器模式；IDAM=01，选择四个 16 位接收过滤器模式；IDAM=10，选择八个 8 位接收过滤器模式；IDAM=11，过滤器关闭。

D2～D0（IDHIT）—过滤器命中指示位。MSCAN 设置该字段来表示标识符接收命中。IDHIT 总是与前台缓冲区（RxFG）的报文相关。当消息被转移到接收器 FIFO 的前台缓冲区中，指示更新。取值范围是 000～111，分别表示过滤器 0～7 命中。

12.3.2 MSCAN 构件封装要点分析

以 MSCAN 驱动构件为例，进行封装要点分析，即分析应该设计哪几个函数及入口参数。MSCAN 具有初始化、发送和接收三种基本操作。按照构件的思想，可将它们封装成三个独立的功能函数，初始化函数完成对 MSCAN 模块的工作属性的设定，接收和发送功能函数则

完成实际的通信任务。对 MSCAN 模块进行编程，实际上已经涉及对硬件底层寄存器的直接操作。另外，发送和接收的内容也需要封装一个函数。

因此，可将初始化、发送和接收三种基本操作所对应的功能函数共同放置在命名为 can.c 的文件中，并按照相对严格的构件设计原则对其进行封装，同时配以命名为 can.h 的头文件，用来定义模块的基本信息和对外接口。

下面就以 MSCAN 模块的初始化、发送和接收三种基本操作为例，来说明实现构件化编程的全过程。

1．初始化函数 CANInit

在初始化函数中主要完成工作模式、过滤模式、时钟源及波特率的配置，所以函数不必有返回值。由于 KEA128 中只有一个 MSCAN 模块，这样初始化函数原型可以设计为：

```
void CAN_Init(uint_8 mode,uint_8 open_filter,uint_32 acc_id);
```

2．填充一个 CAN 数据包 CAN_Fill_STD_Msg

在填充一个 CAN 数据包的函数中主要将节点的 ID、数据长度及数据内容填充到一个数据包结构体变量中。这样该函数原型可以设计为：

```
uint_8 CAN_Fill_STD_Msg(CANMsg *frame,uint_32 id,uint_8 *data,uint_8 len);
```

3．发送 CAN 数据包函数 CAN_Send_Msg

发送 CAN 数据包函数主要用于将填充好的数据包发送出去，因此 CAN 发送数据函数可以设计为：

```
uint_8 CAN_Send_Msg(CANMsg *sendmsg);
```

4．接收 CAN 数据包函数 CAN_Receive_Msg

接收 CAN 数据包函数主要用于接收数据包，因此 CAN 接收数据函数可以设计为：

```
uint_8 CAN_Receive_Msg(CANMsg *recvmsg);
```

12.4　MSCAN 驱动构件的设计

MSCAN 驱动构件存放于网上光盘“..\program\CH12_KEA128_MSCAN”文件夹，供复制使用，各个工程文件夹下的“..\05_Driver\MSCAN”文件夹中 MSCAN 驱动构件与此一致。

12.4.1　MSCAN 初始化

1．MSCAN 初始化过程

（1）使能 MSCAN 模块。

（2）配置 MSCAN 模块进入初始化模式。

（3）配置 MSCAN 模块过滤器与想要接收的标识符 ID。

（4）配置 MSCAN 总线时钟频率及时钟源。

（5）配置 MSCAN 模块工作模式（侦听、回环）。

（6）退出 MSCAN 模块初始化模式。

（7）配置 MSCAN 模块中断。

（8）等待 MSCAN 总线通信时钟同步。

2．MSCAN 初始化函数设计

```
//=============================================================================
//函数名称：CAN_Init
//功能概要：初始化 MSCAN 模块，采用 24MHz 总线时钟作为模块时钟，波特率固定为 500 Kbps
//参数说明：mode:工作模式选择，LOOP_MODE(0)- 回环模式；NORMAL_MODE(1)- 普通模式
//        open_filter:接收过滤器开关
//                  1 - 开启过滤器，只接收标识符 ID 为 acc_id 的帧
//                  0 - 关闭过滤器，接收所有帧
//        acc_id:接收 ID，接收过滤器只接收标识符符合此 ID 的帧，过滤其他 ID 的帧
//函数返回：无
//=============================================================================
void CAN_Init(uint_8 mode,uint_8 open_filter,uint_32 acc_id)
{
    uint_8 i;
    MSCAN_MemMapPtr p = (MSCAN_MemMapPtr) MSCAN_BASE_PTR;

    //MSCAN 的引脚复用选择 CAN_TX 为 PTC7，CAN_RX 为 PTC6
    SIM_PINSEL1 &= (uint_32)～(uint_32)(SIM_PINSEL1_MSCANPS_MASK);
    //MSCAN 模块时钟门控制；SIM_SCGC 的 MSCAN 位=1：使能总线给 MSCAN 时钟；SIM_SCGC
    的 MSCAN 位=0：禁止总线给 MSCAN 时钟
    SIM_SCGC |=(uint_32)(SIM_SCGC_MSCAN_MASK);

    //MSCAN12 模块使能
    p->CANCTL1 |= MSCAN_CANCTL1_CANE_MASK;

    //请求进入初始化模式
    p->CANCTL0 |= MSCAN_CANCTL0_INITRQ_MASK;

    //等待应答进入初始化模式
    while ((p->CANCTL1 & MSCAN_CANCTL1_INITAK_MASK)== 0);

    if(open_filter)                //开启接收过滤器,只接收标示符为 acc_id 的帧
    {
        //使用 2 个 32 位接收过滤器
        p->CANIDAC &= ～MSCAN_CANIDAC_IDAM_MASK;

        //第 1 个 32 位滤波器
        //设置接收寄存器的值
```

```
        p->CANIDAR_BANK_1[0]=acc_id>>3;
        p->CANIDAR_BANK_1[1]=acc_id<<5;
        p->CANIDAR_BANK_1[2]=0xFF;
        p->CANIDAR_BANK_1[3]=0xFF;
        //设置屏蔽寄存器的值
        p->CANIDMR_BANK_1[0]=0x00;
        p->CANIDMR_BANK_1[1]=0x1F;
        p->CANIDMR_BANK_1[2]=0xFF;
        p->CANIDMR_BANK_1[3]=0xFF;

        //第 2 个 32 位滤波器
        //设置接收寄存器的值
        p->CANIDAR_BANK_2[0]=acc_id>>3;
        p->CANIDAR_BANK_2[1]=acc_id<<5;
        p->CANIDAR_BANK_2[2]=0xFF;
        p->CANIDAR_BANK_2[3]=0xFF;
        //设置屏蔽寄存器的值
        p->CANIDMR_BANK_2[0]=0x00;
        p->CANIDMR_BANK_2[1]=0x1F;
        p->CANIDMR_BANK_2[2]=0xFF;
        p->CANIDMR_BANK_2[3]=0xFF;
    }
    else//关闭过滤器
    {
        for (i = 0; i < 16; i++)
        {

            if(i>=0 && i <=3)
                p->CANIDAR_BANK_1[i] = 0xFF;
            else if(i>=4 && i <=7)
                p->CANIDMR_BANK_1[i-4] = 0xFF;
            else if(i>=8 && i <=11)
                p->CANIDAR_BANK_2[i-8] = 0xFF;
            else
                p->CANIDMR_BANK_2[i-12]= 0xFF;
        }
    }

    //配置时钟
    p->CANCTL1 |= MSCAN_CANCTL1_CLKSRC_MASK;     //采用总线时钟(24 MHz)作为模块时钟
    p->CANBTR0 |= MSCAN_CANBTR0_SJW(0);
    p->CANBTR1 |= MSCAN_CANBTR1_SAMP_MASK;      //位时间采样次数为 3
    //MSCAN 传输波特率 = 模块时钟 / [(1 + TSEG1 + TSEG2) * 分频因子]
    //              = 24000K/[(1+16+7)*2]=500Kbps
    p->CANBTR0 |= MSCAN_CANBTR0_BRP(1);         //模块时钟 2 分频
    p->CANBTR1 |= MSCAN_CANBTR1_TSEG2(6);       //TSEG2:6+1=7 Tq
    p->CANBTR1 |= MSCAN_CANBTR1_TSEG1(15);      //TSEG1:15+1=16 Tq
```

```
    //配置工作模式
    if(LOOP_MODE == mode)
        p->CANCTL1 |= MSCAN_CANCTL1_LOOPB_MASK;  //开启回环模式
    p->CANCTL1 &= ~MSCAN_CANCTL1_LISTEN_MASK;   //禁止侦听模式，即开启正常模式

    //退出初始化模式
    p->CANCTL0 &= ~MSCAN_CANCTL0_INITRQ_MASK;   //退出初始化模式，进入正常模式

    //设置中断方式
    p->CANTIER = 0x00;                          //禁止发送中断
    p->CANRIER = 0x01;                          //开启接收满中断
    //设置嵌套向量中断控制寄存器值，使能 CAN 接收中断
    NVIC_ISER |= NVIC_ISER_SETENA30_MASK;

    //等待应答初始化模式
    while ((p->CANCTL1 & MSCAN_CANCTL1_INITAK_MASK) == 1);

    //等待总线通信时钟同步
    while ((p->CANCTL0 & MSCAN_CANCTL0_SYNCH_MASK) == 0);
}
```

12.4.2 MSCAN 发送数据包函数

1．MSCAN 发送数据包过程

CAN_Send_Msg 函数通过 CAN 总线发送数据包，主要执行过程如下。

（1）判断 CAN 总线上的同步状态，确保通信时总线是同步的。

（2）确定 MSCAN 模块中的空闲发送缓冲区以写入数据包。

（3）配置数据包（标识符 ID、数据段、数据长度、发送优先级）。

（4）清 TXE 标志位，发送数据包。

2．MSCAN 发送数据包函数设计

```
//=============================================================================
//函数名称：CAN_Send_Msg
//功能概要：CAN 发送数据包函数
//参数说明：sendmsg:所发送的 CAN 数据包的首地址
//函数返回：0 - 发送成功
//         1 - 数据帧长度错误
//         2 - 发送帧为扩展帧
//=============================================================================
uint_8 CAN_Send_Msg(CANMsg *sendmsg)
{
    uint_8 txEmptyBuf;                          //空闲缓冲区掩码
```

```
uint_8 i;
MSCAN_MemMapPtr p = (MSCAN_MemMapPtr) MSCAN_BASE_PTR;

//检查数据长度
if (sendmsg->m_dataLen > 8)
{
   return 1;                              //数据帧长度错误
}

//检查总线是否同步
while(!(p->CANCTL0 & MSCAN_CANCTL0_SYNCH_MASK));

//寻找空闲缓冲区
txEmptyBuf = 0;
do
{
    p->CANTBSEL = p->CANTFLG;
    txEmptyBuf = p->CANTBSEL;
}
while (!txEmptyBuf);

//写入标识符 ID
if (sendmsg->m_IDE == 0)               //按标准帧填充 ID
{
    p->TSIDR0 = (uint_8)(sendmsg->m_ID>>3);
    p->TSIDR1 = (uint_8)(sendmsg->m_ID<<5);
    p->TSIDR1 |= (sendmsg->m_RTR)<<MSCAN_TSIDR1_TSRTR_SHIFT;
    p->TSIDR1 |= (sendmsg->m_IDE)<<MSCAN_TSIDR1_TSIDE_SHIFT;
}
else
{
   return 2;                              //发送帧为扩展帧
}

//写入数据段
if (sendmsg->m_RTR == 0)               //按数据帧填充
{
   for (i = 0; i < sendmsg->m_dataLen; i++)
   {
       *((&(p->TEDSR[0]))+i) = sendmsg->m_data[i];
   }
   p->TDLR = sendmsg->m_dataLen;
}
else                                   //按远程帧填充
{
    p->TDLR = 0;
}
```

```
    //配置发送优先级
    p->TBPR = sendmsg->m_priority;

    //清 TXEx 标志，通知 CAN 模块发送
    p->CANTFLG = txEmptyBuf;

    return 0;                                   //配置发送完成
}
```

12.4.3 MSCAN 接收数据包函数

1．MSCAN 接收数据包过程

CAN_Receive_Msg 函数完成从接收缓冲区读取数据包的功能，主要执行过程如下。

（1）检测接收状态标志位 RXF，若没有数据退出。

（2）判断帧的类型（标准帧/扩展帧、数据帧/远程帧），以进行相应的处理。

（3）处理标准数据帧，读数据包中数据段内容。

（4）清 RXF 标志位，释放接收缓冲区。

2．MSCAN 接收数据包函数设计

```
//=============================================================================
//函数名称：CAN_Receive_Msg
//功能概要：接收 CAN 数据包函数
//参数说明：receiveFrame:存放待接收 CAN 数据包缓冲区的首地址
//函数返回：0 - 成功接收标准帧
//         1 - 未接收到帧
//         2 - 收到扩展帧
//=============================================================================
uint_8 CAN_Receive_Msg(CANMsg *recvmsg)
{
    uint_8 i;
    MSCAN_MemMapPtr p = (MSCAN_MemMapPtr) MSCAN_BASE_PTR;

    //检测接收标志
    if( (p->CANRFLG & 0x01) == 0)           //(CAN0RFLG_RXF == 0)
    {
        return 1;                           //未接收到帧
    }

    //判断标准帧/扩展帧
    if( (p->RSIDR1 & 0x04)  == 0)           //收到标准帧
    {
        recvmsg->m_ID = (uint_32)(p->RSIDR0<<3)
                      | (uint_32)(p->RSIDR1>>5);
        recvmsg->m_RTR = p->RSIDR1 & 0x08;
        recvmsg->m_IDE = 0;
    }
```

```
    else                                        //收到扩展帧
    {
        return 2;
    }

    //判断数据帧/远程帧
    if((p->RSIDR1 & 0x08) == 0)                 //收到数据帧
    {
        recvmsg->m_dataLen = p->RDLR;           //数据长度
        //读取数据段内容
        for (i = 0; i < recvmsg->m_dataLen; i++)
        {
            recvmsg->m_data[i] = *((p->REDSR)+i);
        }
    }
    else                                        //收到远程帧
    {
        recvmsg->m_dataLen = 0;
    }

    //清 RXF 标志位，准备接收下一帧
    p->CANRFLG = 1;

    return 0;                                   //成功接收标准帧
}
```

第 13 章

系统时钟与其他功能模块

本章导读：本章主要介绍基本功能模块外的其他功能模块，主要内容有：（1）系统时钟的概述与设置；（2）复位模块；（3）看门狗模块；（4）电源模式与芯片配置；（5）循环冗余检查、杂项控制模块及交叉开关。这些内容一般会在程序的初始化中使用。与前面的章节相比，此章内容较复杂且难理解，但对于 KEA128 芯片的开发，又是必不可少、需要全面理解的部分。

本章参考资料：本章 13.1 节（系统时钟模块）总结自《KEA128 参考手册》的第 5 章；13.2 节（复位模块）总结自《KEA128 参考手册》的第 6 章；13.3 节（看门狗模块）总结自《KEA128 参考手册》的第 16 章；13.4 节（电源模式与芯片配置）总结自《KEA128 参考手册》的第 3 章和第 7 章；13.5 节（循环冗余检查、杂项控制模块及交叉开关）总结自《KEA128 参考手册》的第 14 章，第 22 章。



13.1 时 钟 系 统

时钟系统是 MCU 不可或缺的组成部分，就像人的心跳一样，它产生的时钟信号要贯穿整个芯片。KEA128 的时钟系统比较复杂，不像 51 单片机那样仅采用一个系统时钟。同一个电路，时钟越快功耗越大，同时抗电磁干扰能力也会越弱。也并非所有的外设都需要那么高的工作频率，例如，给看门狗提供 1 kHz 的工作时钟即可正常运行，所以对于较为复杂的 MCU 一般都是采取多时钟源的方法来解决这些问题的。

13.1.1 时钟系统概述

KEA128 芯片的时钟系统由系统振荡器（System oscillator，OSC）、内部时钟源（Internal Clock Source，ICS）、低功耗振荡器（Low-Power Oscillator，LPO）、系统集成模块（SIM）和电源管理器（Power Management Controller，PMC）组成。OSC 由一个外部晶振向 MCU 提供参考时钟，ICS 包括一个内部参考时钟（ICSIRCLK）和锁频环（FLL），LPO 使用 PMC 提供的 1 kHz 的低功耗晶振，基于这一独立的低频时钟源，MCU 运行在所有的电源模式下，也可为 RTC 和看门狗（WDOG）提供参考时钟，SIM 模块为各模块选择工作所需的时钟源，PMC 模块可提供 1 kHz 的参考时钟信号。时钟系统的框图如图 13-1 所示。

OSC 实际上是 3 个外部时钟源组合成 1 个，它包含一个使用 31.25～39.0625 kHz 晶体振荡器或陶瓷振荡器的低频振荡器，以及一个使用 4～24 MHz 晶体振荡器或陶瓷振荡器的高频振荡器，每个振荡器都需要两个引脚 XTAL 和 EXTAL 来产生时钟信号。OSC 还有一种外部

时钟模式，只允许一个外部时钟信号进入 MCU。在该模式下，只需要 EXTAL 引脚，XTAL 引脚可作为普通 I/O 口用。输入频率可以是 0 Hz～24 MHz 间的任意值。由于引脚限制，部分 MCU 可以没有外部振荡器，如在本评估板上并没有采用外部振荡器。

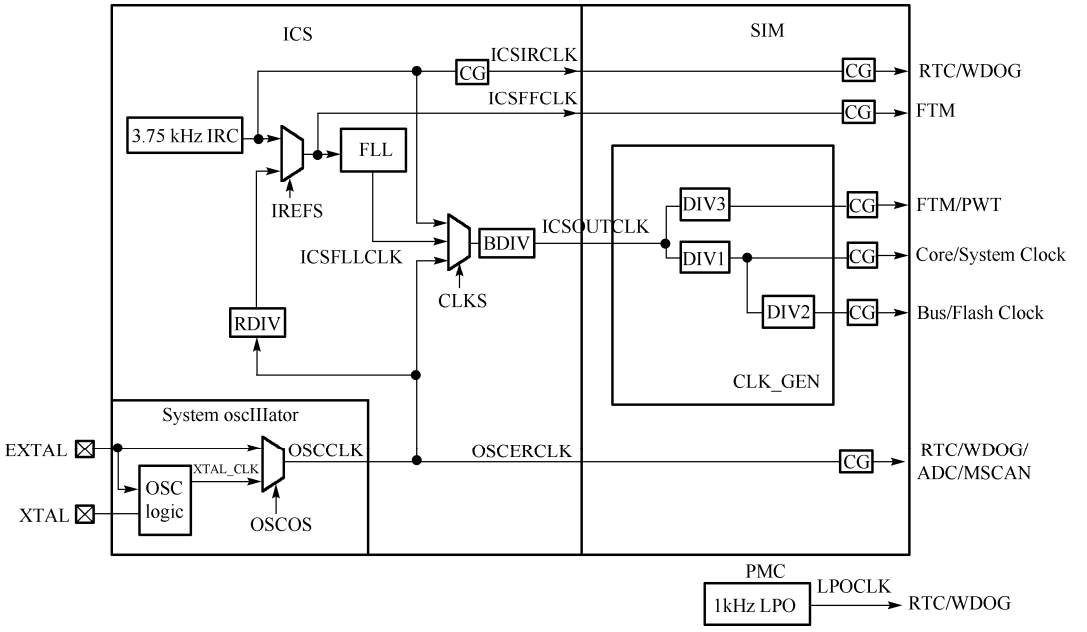


图 13-1 时钟系统框图

在不使用外部振荡器的情况下，KEA128 芯片的时钟源由 ICS 模块和 LPO 模块提供。ICS 模块主要由内部参考时钟（Internal Reference Clock，IRC）和锁频环（Frequency Lock Loop，FLL）两个子模块组成。由于工厂处理的不同，IRC 的频率必须在 31.25 kHz～39.0625 kHz 范围内，本芯片的缺省 IRC 频率是 37.5 kHz，通过软件设置 ICS 控制寄存器 ICS_C3 的所有 8 位和 ICS_C4 的最低位可进行校准与修改，缺省频率对应的寄存器 ICS_C3_SCTRIM=0x69，ICS_C4_SCTFRIM=0。FLL 非常类似于锁相环（PLL），区别在于 PLL 基于参考时钟和数控振荡器（Digital Controlled Oscillator，DCO）时钟的相位差调节输出，而 FLL 通过比较 DCO 时钟和参考时钟的频率来工作。本芯片的 FLL 的输出频率是输入参考时钟频率的 1280 倍。例如，IRC 为 37.5 kHz，那么 FLL 的输出频率 ICSFLLCLK 为 37.5 KHz×1280=48 MHz。

系统中主要的时钟来自于 ICSFLLCLK 经 BDIV 分频的 ICSCLKOUT，用于产生 CPU 和总线时钟的时钟信号。ICS 的一种配置是，CPU 时钟频率等于 ICSCLKOUT 的 1/2，总线时钟频率为 ICSCLKOUT 的 1/2，即 BDIV=1，DIV1=0，DIV2=0，其中，数字 0 代表 1 分频，即 ICSCLKOUT/1，相应地数字 1 代表 2 分频。

ICS 与 ICG（Internal Clock Generator，内部时钟发生器）相比，是一个全新的模块，但在功能上基本是 ICG 的一个子集。ICS 的用途是为更小、更低成本的 MCU 提供一个更小的低功耗的时钟模块。一个重要的区别在于，ICS 中的 FLL 固定用 1280 的倍频因子，如果输入范围是 31.25 kHz～37.5 kHz，则产生的输出频率为 40 MHz～48 MHz，而 ICG 中的 FLL 有 8 个软件可选择的倍频因子，外加一个同时使用高频（> 4MHz）和低频（<100 Hz）时钟参考的预分频器，产生的输出频率为 4 MHz～48 MHz，且对参考时钟的限制更少。

13.1.2　时钟模块概要与编程要点

各路时钟的选择和复用是通过 ICS 模块来编程控制的，而系统的时钟分频器和模块时钟门是通过 SIM 模块来编程设置的。

KEA128 的 ICS 模块共有 7 种工作模式：FEI、FEE、FBI、FBILP、FBE、FBELP 和 STOP。表 13-1 给出了各种工作模式的含义。

表 13-1　ICS 工作模式

STOP	停止模式。只要MCU进入低功耗STOP模式，ICS就被关闭，不再向MCU提供任何一路时钟。在STOP模式下，FLL被禁止以节省功耗
FEI	FLL内部参考模式。芯片复位后的默认模式，包括上电复位（POR）。在该模式下， FLL被激活且由其输出时钟信号驱动CPU和总线等。IRC被用作FLL的参考频率，所以不需要外部元件（如过滤电容或校准电阻），EXTAL和XTAL引脚可用于其他复用功能
FEE	FLL外部参考（FEE）模式。在FLL 使用外部参考（FEE）模式下，FLL被激活，其输出驱动CPU和总线。外部振荡器为FLL提供参考时钟。为了保证系统的正常工作，FLL输出必须介于40～48 MHz之间。因此，使用FEE模式对外部参考的频率有一定的限制。包含一个ICS的参考分频器用于降低外部参考的频率，以提供更多的外部时钟选项。参考分频器由ICS控制寄存器1（ICSC1）的RDIV位段确定
FBI与FBILP	FLL旁路内部参考（FBI）模式与FLL 旁路内部参考低功耗（FBILP）模式。FLL旁路内部参考（FBI）模式和FLL 旁路内部参考低功耗（FBILP）模式只有一个不同点。它们都是绕过FLL直接使用IRC产生ICSFLLOUT。不同之处在于FBI模式下FLL仍是激活的，而在FBILP模式下FLL被禁止了，所以可以节省更多的电源。通过配置成FBI并设置ICS 控制寄存器2（ICSC2）的LP位可以使能FBILP
FBE与FBELP	FLL旁路外部参考（FBE）模式与FLL旁路外部参考低功耗（FBELP）模式。FLL旁路外部参考（FBE）模式和FLL旁路外部参考低功耗（FBELP）模式与上面所述的FBI和FBILP模式相类似，不同在于是用OSC产生时钟源，而不是IRC。FBE/FBELP也绕过了FLL，可通过LP位来禁止FLL

对于时钟的配置，必须使各路时钟满足一定的要求，表 13-2 提供了一种推荐的时钟配置。

表 13-2　推荐时钟配置

时　钟	频　率
内核/系统	24 MHz
总线	24 MHz

（1）内核和系统时钟频率必须在 48 MHz 以内。

（2）总线时钟频率不大于 24 MHz，且必须使内核时钟的整数分频。

就 KEA128 的配置要点，总结如下。

（1）使能时钟源，如 IRC，外部振荡器。

（2）如果采用 IRC，需设置 IRC 的时钟频率；若采用外部振荡器，需设置参考分频器 RDIV 的值，使得分频后的频率范围控制在 31.25 kHz～39.0625 kHz 之间，以满足 FLL 的输入时钟频率要求。

（3）等待 FLL 时钟输出稳定。

（4）配置内核及总线时钟等，包括 BDIV、DIV1、DIV2。

13.1.3　时钟模块测试实例

时钟初始化过程代码如下所示。

```
//初始化时钟配置（采用内部参考时钟，系统时钟 48 MHz，总线时钟 24 MHz）
#if (CLOCK_SETUP == 0)
```

```
//采用 24MHz 的 FLL 输出作为时钟源
ICS->C2 |= ICS_C2_BDIV(0x01);
//更新系统时钟分频值
SIM->CLKDIV = SIM_CLKDIV_OUTDIV1(0x00) |   //将 ICSOUTCLK 作为系统时钟
              SIM_CLKDIV_OUTDIV2_MASK |    //将系统时钟的 1/2 作为总线时钟
              SIM_CLKDIV_OUTDIV3_MASK;     //将系统时钟的 1/2 作为定时器时钟
//切换到 FEI 模式
ICS->C1 = ICS_C1_CLKS(0x00) |//选中 FLL 作为时钟源
          ICS_C1_IREFS_MASK |//选择 37.5 kHz 的内部参考时钟作为 FLL 的参考时钟源
          ICS_C1_IRCLKEN_MASK;//使能内部参考时钟
//将 FLL 输出直接作为 ICSOUTCLK 的值，此时 ICSOUTCLK=37.5 kHz*1280=48 MHz
ICS->C2 = (uint8_t) ((ICS->C2 & (uint8_t) ～(uint8_t) (
ICS_C2_BDIV(0x07) | ICS_C2_LP_MASK)) | (uint8_t) (ICS_C2_BDIV(0x00)));
//等待选中内部参考时钟作为 FLL 的参考时钟源
while ((ICS->S & ICS_S_IREFST_MASK) == 0x00U) {}
//等待选中 FLL 输出作为时钟源
while ((ICS->S & 0x0CU) != 0x00U){}
```

13.2 复 位 模 块

从引起 MCU 复位的内部与外部因素来区分，复位可分为外部复位和内部复位两种。外部复位有上电复位、按下“复位按钮”复位；内部复位有看门狗定时器复位、低电压复位、低漏唤醒（Low Leakage Wake-up Unit，LLWU）复位等、软件复位等。

从复位时芯片是否处于上电状态来区分，复位可分为**冷复位和热复位**。芯片从无电状态到上电状态（Power-On Reset，POR）的复位属于冷复位，芯片处于带电状态时的复位叫热复位。冷复位后，MCU 内部 RAM 的内容是随机的；而热复位后，MCU 内部 RAM 的内容会保持复位前的内容，即热复位并不会引起 RAM 中内容的丢失。

从 CPU 响应快慢来区分，复位还可分为异步复位与同步复位。异步复位源的复位请求一般表示一种紧要的事件，因此复位控制逻辑不等到当前总线周期结束，复位立即有效。异步复位源有上电、低电压复位等。同步复位的处理方法与异步复位不同，当一个同步复位源给出复位请求时，复位控制器并不使之立即起作用，而是等到当前总线周期结束之后，这是为了保护数据的完整性。在该总线周期结束后的下一个系统时钟的上升沿时，复位才有效。同步复位源有看门狗定时器、软件等。

芯片被正确写入程序后，经复位或重新上电后才可启动执行程序。当出现异常时，也可通过复位，使得芯片回复到最初已知状态，以对系统进行保护。KEA128 支持的复位源见表 13-3。

表 13-3 复位源

复 位 源	描 述
上电复位	上电复位（POR）
系统复位	外部引脚（PIN）复位；低电平检测（LVD）复位；看门狗（WDOG）复位；多用途时钟发生器时钟丢失（LOC）复位；停止模式应答错误（SACKERR）复位；软件复位（SW）；锁定复位（LOCKUP）；MDM-DAP 系统复位
调试复位	复位调试子系统

13.2.1　上电复位

当给 MCU 上电或提供的电压低于上电复位重置电压（V_{POR}）时，POR 电路会触发 POR 复位。

当电压升高时，LVD 电路保持 MCU 处于复位状态直到电压大于 LVD 低电压阈值（V_{LVDL}）。POR 复位后 SRSL 寄存器的 POR 和 LVD 位也要重置。

13.2.2　系统复位源

MCU 复位是一种可以使芯片回到初始状态的方法。系统复位从完全稳定的稳压器产生系统时钟由内部参考时钟产生。当芯片退出复位时，它执行以下操作。

- 从中断向量表偏移 0 读取起始 SP（SP_main）；
- 从中断向量表偏移 4 读取起始 PC；
- LR 设置为 0xFFFF_FFFF。

片上外设模块和非模拟 IO 引脚初始化都被置为禁用（根据“系统选择寄存器（SIM_SOPT）的设置”章节，系统复位后，SWD_DIO/SWD_CLK、$\overline{\text{NMI}}$ 和 $\overline{\text{RESET}}$ 引脚可用）。复位之后模拟引脚被默认为相应的模拟功能。

1．外部引脚复位（RESET）

该引脚可以内部上拉，复位后，将芯片从任何模式唤醒。

POR 置位后，PTA5 引脚复用为 RESET 功能。SIM_SOPT[RESET]必须编程才能使能其他功能。当该字段被清 0，该引脚与 PTA5 或其他可选功能相同。

2．低电平检测（LVD）复位

该芯片包含一个系统来防止低压条件，是为了在电源电压的变化时保护存储内容和单片机控制系统状态。这个系统包括一个上电复位（POR）电路和一个 LVD 电路，这个 LVD 电路具有可选的跳闸电压，高（V_{LVDH}）或低（V_{LVDL}）。

设置 PMC_SPMSC1[LVDE]后，LVD 电路可用，跳闸电压是由 PMC_SPMSC2[LVDV]选择的。进入 Stop 模式，LVD 禁用，除非设置 PMC_SPMSC1[LVDSE]或在 SWD 模式。在 Stop 模式下，如果同时设置 PMC_SPMSC1[LVDSE] 和 PMC_SPMSC1[LVDE]，电流消耗会增加，同时激活 LVD。

3．COP 看门狗复位

看门狗定时器（WDOG）通过来自软件的预期周期性通信，来监控系统的操作，这种通信通常称为服务（或刷新）看门狗。如果不发生周期刷新，看门狗会引起系统复位。WDOG 重置会引起设置 SIM_SRSID [WDOG]。

4．多功能时钟发生器时钟丢失（LOC）复位

ICS 支持外部参考时钟监控，并具有重置功能。

在 FBE 和 FEE 模式，如果向 ICS_C4[CME]写 1，使能时钟监控。如果外部参考时钟频率低于一个确定的频率。例如，取决于 OSC_CR[RANGE]的 f_{loc_high} 或 f_{loc_low}, MCU 会重置。设置 SIM_SRSID[LOC]以显示错误。

在 FBILP 模式，FLL 关闭，所以即使向 ICS_C4[CME]写 1，外部参考时钟监控也不起作用。

外部参考时钟监控使用 FLL 作为内部参考时钟。设置 ICS_C4[CME]之前，FLL 必须可用。

5．停止模式应答错误（SACKERR）复位

如果内核试图进入 Stop 模式，该复位产生，但是不是所有模块的 Stop 模式在 1 kHz 的 LPO 时钟的 1025 个周期内。

如果一个错误产生，该模块可能并不应答进入 Stop 模式，该错误也可能由外部时钟输入一个模块中产生。

6．软件复位（SW）

NVIC 应用中断和复位控制寄存器的 SYSRESETREQ 位置位时会产生一个软件复位（参见 ARM 的 NVIC 文档寄存器详细说明，尤其 VECTKEY 位域要求说明）。置位 SYSRESETREQ 可产生软件复位请求。除了调试模块，软件复位能重设系统大多数模块。

7．锁定复位（LOCKUP）

LOCKUP 会立即表明严重错误的内核软件产生的错误，在内核中系统硬件保护激活时，一个不可恢复异常导致内核被锁。

LOCKUP 导致系统复位，也可使 SIM_SRSID[LOCKUP]置位。

8．MDM-DAP 系统复位

通过设置 MDM-DAP 控制寄存器中的系统复位请求位可以产生系统复位。该方法是通过 SWD 接口的复位的主要复位方法，直到该位被清除，系统复位才停止。

当芯片从系统复位中唤醒，通过设置 MDM-DAP 控制寄存器中的内核保持复位位可以控制正在复位的内核。

13.2.3 调试复位

MCU 不同模块产生各种不同的复位。

1．POR Only

POR Only 复位只在上电复位出现时可用，它可以复位 PMC 和 RTC，POR Only 也可以引起其他类型的复位。

2．Chip POR

芯片上电复位只在 POR 和 LVD 复位源出现时可用，它可以复位复位引脚过滤寄存器和 SIM、ICS 的一部分，芯片上电复位也可以引起芯片复位（包括早期芯片复位）。

3．Early Chip Reset

早期芯片复位在所有的复位源上可用，它只复位 Flash 存储模块和 ARM，在 Flash 初始化之前，不产生早期芯片复位。

4．芯片复位

芯片复位在所有的复位源上可用，并且只在 $\overline{\text{RESET}}$ 引脚不可用后芯片复位消失，它复位其他模块（其他类型复位不能复位这些模块）。



13.3　看　门　狗

13.3.1　功能描述

看门狗定时器（WDOG）的全称为 Computer Operating Properly（COP）Watchdog，也可以简称 COP。看门狗定时器具有监视系统功能，当运行程序跑飞或一个系统中的关键系统时钟停止引起严重的后果的情形下，无法回到正常的程序上执行，看门狗通过复位系统的方式，将系统带到一个安全操作的状态。正常情况下，看门狗通过与软件的定期通信来监视系统的执行过程，清看门狗计时器，即定期喂看门狗。如果应用程序丢失，未能在看门狗计数器超时之前复位，则将产生看门狗复位，强制将系统恢复到一个已知的起点。任何复位后，看门狗都将被使能。如果应用程序不使用看门狗，可以通过清 WDOG_CS1[EN]位来禁用看门狗。

13.3.2　配置 WDOG

所有的 WDOG 控制位，超时值和窗口值在复位之后只写一次，这意味着写发生后它们不能更改（除非发生复位或解锁操作）。这提供了强健的机制来配置 WDOG，并且保证了在配置后，跑飞情况不会错误地禁止或修改 WDOG 配置。

（1）解锁 WDOG。由于 WDOG 模块各寄存器多数都是只写一次位的，所以在重新配置 WDOG 之前，都要进行一个解锁序列操作。解锁序列是指，在 WDOG 配置后的 16 个总线时钟周期内，写 0x20C5 和 0x28D9 到 WDOG_CNTH:L 寄存器。完整的解锁序列要求用户必须在 128 个总线时钟内重新配置 WDOG；否则，WDOG 强制复位 MCU。

（2）刷新 WDOG。刷新写序列是对 WDOG_CNTH:L 寄存器先写 0x02A6 后写 0x80B4。必须在写 0x02A6 后的 16 个总线时钟内，写 0x80B4，即喂狗操作，从而防止 WDOG 复位；否则，WDOG 复位 MCU。

注意：在开始刷新序列之前，禁用全局中断；否则，如果写 4 个字节超过 16 个总线时钟，一次中断可以使刷新序列失效，当序列完成时重新启动中断。

13.3.3 测试实例

测试实例位于网上光盘中的“..\ program\CH13-KEA128-WDOG”文件夹。在系统开发过程中，一般先关闭看门狗的功能，避免不必要的复位启动时间的发生（见 sysinit.c）。只有在系统开发完成，调试正常准备投入使用时，才开启看门狗的功能。

13.4 电源模式与芯片配置

13.4.1 电源模式控制

PMC 提供多种可选电源，用户可以根据不同的功能需求来选择不同的模式以优化电源功耗。该设备支持 Run、Wait 和 Stop 模式，用户可以根据不同的功耗水平和功能需求选择不同的电源模式和功耗模式。I/O 状态在所有模式操作中都会保留。

（1）Run 模式—CPU 可以运行在全速模式，内部电源完全监管。

（2）Wait 模式—CPU 减速以节约电量；系统时钟和总线时钟持续工作，并且保持完全监管。

（3）Stop 模式—低电压指令（LVD）选项使能，电压调节器处于待命状态。

CPU 有三种基本模式：运行、等待和停止。WFI 指令可以进入等待和停止模式，芯片通过运行、等待和停止三种模式的不同排列来实现低功耗，见表 13-4。

表 13-4 芯片电源模式

电 源 模 式	描 述	内 核 模 式	正常恢复方法
正常运行	允许芯片功能最大化。复位默认模式，片上芯片稳压器打开	Run	—
正常等待，通过WFI指令	当芯片处于睡眠模式时，允许外设正常工作，降低功耗。NVIC仍然响应中断，外设持续使用时钟	Sleep	中断
正常停止，通过WFI指令	使芯片处于静止状态，极低功耗模式保持所有的寄存器状态，同时保持LVD保护功能。NVIC功能被禁用；AWIC被用于从中断唤醒；外设时钟禁用	Sleep Deep	中断

13.4.2 低功耗下的模块操作

在 Run 模式下，各个功能模块都处于运行状态；在 Wait 模式下，CPU 切换到待机状态；在 Stop 模式下，各功能模块都转为待机或者选择性运行。详细内容请参考《KEA128 参考手册》第 7 章。

13.4.3 芯片配置模块

本节给出微控制器内部模块的配置信息，主要包括 9 个部分的配置。

（1）内核模块：基于 ARM Cortex-M0+的 32 位 MCU 内核、嵌套中断向量表（NVIC）的配置、异步唤醒中断控制器（AWIC）的配置等。

（2）系统模块：系统集成模块（SIM）、电源管理（PMC）、杂项控制模块（MCM）、位操作引擎（BME）、交叉开关、外设桥（AIPS-Lite）配置。

（3）系统安全功能配置：主要涉及循环冗余检查（CRC）、具有独立时钟源的看门狗（WDOG）配置等。

（4）时钟模块配置：主要涉及 ICS 配置、OSC 配置及锁频环 FLL。

（5）存储和存储接口：128 KB 的 Flash 内存及其主控器、16 KB 的 SRAM 及其主控器等。

（6）模拟量相关部分配置：16 个通道的 12 位模/数转换器（ADC）、比较器（ACMP）及可选时钟等。

（7）定时器模块：Flex 定时器（FlexTimer）、定时器（PIT）、实时计数器（RTC）及脉冲宽度定时器（PWT）、系统节拍定时器（SysTick）等。

（8）通信接口模块：主要涉及 SPI、IIC、UART、MSCAN 等通信接口。

（9）人机接口模块（HMI）：主要涉及 GPIO 配置、KBI 配置及 IRQ 配置等。

通过对这 9 个方面的配置，使得芯片在运行时能够启动相应的模块功能，并设置详细的工作参数，详细信息参见本书各不同章节模块介绍。

13.5　循环冗余检查、杂项控制模块与交叉开关

13.5.1　循环冗余检查

循环冗余检查（Cyclic Redundancy Check，CRC）具有数据传输检错的功能。一方面，发送设备首先对所传数据进行多项式计算，将得到的结果附在帧的后面；另一方面接收设备也进行类似的操作，从而确保数据传输的正确性和完整性。KEA128 芯片的循环冗余校验模块产生 16/32 位的循环冗余校验码，用于错误检测。

13.5.2　杂项控制模块

杂项控制模块（Miscellaneous Control Module，MCM）提供丰富的控制功能。例如，对 Flash 模块相关的操作，可参考 Flash 相关章节。

MCM 包含下述特征。

（1）关于平台配置的可视化编程信息。

（2）Flash 控制器预读与 Cache 配置。

13.5.3　交叉开关

交叉开关（Crossbar Switch）使用交叉开关结构连接总线主机和总线从机。这样的结构允许所有总线主机同时访问不同的总线从机，当访问同一个从机时，多个主机之间有一个仲裁机制。所有总线仲裁方法及属性是基于从机进行编程的。

1．功能说明

当主机访问交叉开关时，访问被立刻处理。若目标从机端口可访问，则立刻访问从机端口；若访问的目标从机端口正忙或被另一个主机停止，请求主机等待直到目标从机端口可以处理主机的请求。

在主机获得目标从机端口的控制权后，主机保持对从机端口的控制，直到运行一个空闲周期或在下一次访问时离开当前从机端口而放弃对从机端口的控制。若另一个更高优先级主机产生请求访问此从机端口，主机就失去对从机端口的控制。但是，若主机运行一个固定长度的突发传输，它将保持对从机端口的控制，直到传输结束。

2．仲裁

交叉开关支持两种仲裁机制：固定优先级比较算法和轮询公平算法，每个从机端口的仲裁机制是独立编程的。

第 14 章

KEA128 在实时操作系统 MQX-Lite 下的应用

本章导读：本章主要阐述有关 KEA128 在 MQX-Lite 下的编程设计。本章 14.1 节简要概述了 MQX-Lite 的定义及其组成部分；14.2 节给出 MQX-Lite 编程知识要素，包括任务管理与调度、任务间同步与通信及中断处理机制等，为编程打下理论知识基础；14.3 节给出基于 MQX-Lite 的 KEA128 工程框架，以达到规范编程的目的；14.4 节给出 KEA128 在 MQX-Lite 下的第一个样例工程。期望通过本章的学习，能够基本掌握实时操作系统 MQX-Lite 下的程序设计。

本章参考资料：本章主要参考《嵌入式实时操作系统 MQX 应用开发技术—ARM Cortex-M 微处理器》、MQX-Lite 用户手册和参考手册。

14.1 MQX-Lite 简介

MQX-Lite 实时操作系统是标准 MQX[①]（Message Queue eXecutive，消息队列执行）实时操作系统的轻量级版本，在组成与功能上都是标准 MQX 的子集，主要针对资源有限的 MCU，支持应用以低于 4 KB 的 RAM 空间运行。

MQX-Lite 并不随标准的 MQX 一起独立发布，而是打包作为处理器专家（Processor Expert，PE）的一个组件提供，目前的最新版本为 MQX-Lite V1.1.1。

MQX-Lite 由 MQX-Lite 内核与 PSP（处理器支持包）组成，表 14-1 列举出了 MQX-Lite 内核的所有组件及其功能。

表 14-1 MQX-Lite 内核组件

类　型	名　称	内容及作用
核心组件	初始化	初始化及创建自启动任务
	任务管理	任务的创建、管理和终止
	调度	基于优先级的抢占式 FIFO 任务调度
	轻量级信号量	实现轻量级信号量同步机制
	中断和异常处理	处理所有的硬件中断
可选组件	轻量级事件	实现轻量级事件同步机制
	互斥量	实现互斥量同步机制
	轻量级消息队列	实现任务间通信
	轻量级定时器	用于定时或周期性地调用应用函数
	内核日志	记录 MQX-Lite 的活动
	轻量级存储分配	动态存储分配

① 关于标准版 MQX 实时操作系统的使用，可以参考本文作者撰写的《嵌入式实时操作系统 MQX 应用开发技术》。

由表 14-1 可知，使用 MQX-Lite 实时操作系统，可以实现诸如多任务处理、基于优先级的抢占式任务调度、共享资源的同步访问、任务间的通信及中断处理等功能。

14.2 MQX-Lite 编程知识要素

从程序编写的角度上来说，使用 MQX-Lite 实时操作系统需要了解任务管理与调度、任务间同步与通信及中断处理机制等知识要素。

14.2.1 任务管理与调度

1．任务的基础知识

在无操作系统下进行编程时，一般会将程序的功能都放在主函数中按一定顺序执行；而在操作系统下的编程，则是将这些程序功能划分为多个任务，在主函数中只需启动操作系统，之后由操作系统负责调度这些不同功能的任务。

任务是完成一定功能的函数，但是并不是所有的函数都可以被称为任务。从任务的存储结构上看，MQX-Lite 的任务由三个部分组成，即**任务的三要素：任务函数、任务堆栈、任务描述符**。任务函数就是任务要完成具体功能的程序。每个任务拥有自己独立的任务栈空间，用于保存任务在调度时的上下文信息及任务内部使用的局部变量。任务描述符关联了任务代码的程序控制块，记录任务的各个属性。

2．任务管理

MQX-Lite 中提供了多种任务管理功能，下面只对其中最常用的两个功能（任务创建与阻塞）进行介绍。

1）任务的创建

在创建任务前，需要将设计好的任务函数在“任务模板”（Task Templete）中进行登记。所有的任务均以任务模板为依据进行创建，任务模板包含了创建任务所需要的信息，如任务编号、任务函数、任务栈大小、优先级、任务名、任务属性等。

MQX-Lite 维护了一个缺省的任务模板列表 MQX_template_list[]，供用户添加应用任务模板，在任务模板列表中至少有一个任务模板具有自启动任务属性，MQX-Lite 在启动过程中会查询任务模板列表，创建并启动其中的自启动任务。而对于其他的任务，就需要调用任务创建函数在程序中创建了。

由于 MQX-Lite 不支持任务的动态创建，所以不能使用标准 MQX 中的_task_create()函数来创建任务。取而代之的是使用_task_create_at()函数，利用已经静态分配好的任务栈空间来创建任务。创建好的任务将立刻加入到就绪任务队列，等待分配 CPU 使用权即可运行。

一般地，通过在自启动任务中调用 task_create_at 函数的方式，将应用系统的其他任务创建出来。

2）任务的阻塞

通过调用_task_block 函数可以阻塞当前运行的任务，将当前任务的状态属性 TD_STATES 设置为 BLOCK，并且将该任务从就绪队列移除，不回收资源，直接进入上下文切换，调度当前就绪任务队列优先级最高的任务来执行。被阻塞的任务仍保留在系统中，若没有其他任务主动调度，该任务将不再运行。

一般可以在自启动任务执行完后调用_task_block 函数将其阻塞。

3．任务调度

MQX-Lite 中使用基于优先级抢占机制的先进先出（First In First Out，FIFO）调度策略。MQX-Lite 中的任务都以优先级数表示优先级，优先级数越小，优先级越高。

基于优先级抢占的调度策略是指在运行时，MQX-Lite 总是将 CPU 的使用权分配给当前就绪的、优先级最高的任务。具体来说分三种情况：

（1）设任务 B 的优先级高于任务 A，当任务 A 正在运行时，任务 B 准备就绪（发生情景：1 任务 A 创建了任务 B；2 任务 B 的延时到期，则调度系统在下一个 Systick 中断发生的时候将 CPU 的使用权从任务 A 处抢夺，将其转入就绪态，并分配 CPU 使用权给任务 B。

（2）当任务 A 被阻塞后主动放弃 CPU 使用权，此时，调度系统将选择当前就绪的任务中寻找优先级最高的任务，将 CPU 的使用权分配给它。

（3）当存在同一优先级的多个任务都处于就绪态时，较早进入就绪态的任务优先获得调度的机会。

当发生以下任意一种情况时，当前任务停止运行。

（1）由于调用了 MQX-Lite 阻塞功能函数或延时函数，活动任务主动放弃处理器使用权。

（2）产生了一个比活动任务所能屏蔽的中断优先级更高的中断。

（3）更高优先级的任务已经处于就绪状态。

其中延时函数的说明：在 MQX-Lite 的任务函数中，一般不使用原地空跑的方式进行延时，而是调用_time_delay_ticks 延时函数，通过使用延时任务队列管理延时任务来实现对任务的延时。

例如，正在运行的任务调用_time_delay_ticks 函数，如_time_delay_ticks（30）（其中实参 30 表示延时 30 个时钟滴答），该函数执行时，会将当前系统时刻与传入延时时间参数相加，得到该任务延时后应该唤醒的时刻，然后将该任务从就绪任务队列中撤出，以唤醒的时刻插入延时任务队列中。这样，当延时的时间到达时，可以执行延时函数后面的程序，延时期间，该任务已经放弃 CPU 的控制权，内核调度正常进行，可以执行别的就绪任务。

14.2.2　任务间同步与通信

为了协调不同任务间的执行顺序及在不同任务间传输数据，在 MQX-Lite 中提供了诸如轻量级事件、轻量级消息队列、轻量级信号量、互斥量等方式来实现任务间的同步与通信。

1．轻量级事件

访问一个共享数据区、完成一组数据的处理等可以被系统识别的操作都可以看成一个 RTOS 中的事件（Event）。一个事件用一位二进制数（0/1）表达。在 MQX-Lite 中，通常用一个字（如 32 位）来表达事件，这个字被称为事件组（Event Group）。事件组每一位可记录一个事件（也称为事件位），且事件之间相互独立，互不干扰。

标准版 MQX 中提供了两种类型的事件：轻量级事件（Lightweight Event）和事件（Event）。而在 MQX-Lite 中只有轻量级事件。

1）轻量级事件特点

轻量级事件（Lightweight Event）是 MQX-Lite 的可选组件，其管理机制和使用操作相比事件更为简单，且具有处理速度快、占用内存空间少等优点，主要用于对系统的执行效率有比较严格要求的应用场合。

2）轻量级事件常用函数

轻量级事件的常用操作有创建、置位、等待、清位等，表 14-2 给出了这些常用操作函数的信息。至于其他轻量级事件的操作函数说明，请参见 MQX-Lite 的参考手册。

表 14-2 轻量级事件操作函数

序 号	函 数 名 称	函 数 功 能
1	_lwevent_create	创建轻量级事件组
2	_lwevent_set	置 1 轻量级事件位
3	_lwevent_wait_ticks	等待轻量级事件位（最长等待时间的单位为时间滴答数）
4	_lwevent_clear	清 0 轻量级事件位

3）轻量级事件使用方法

轻量级事件组件是 MQX-Lite 的可选组件，在首次使用前需在“..\config\user_config.h”文件中，配置启用轻量级事件组件，代码如下。

```
#define MQX_USE_LWEVENTS 1   //轻量级事件组件的启用与禁用（1-启用，0-禁用）
```

轻量级事件是全局结构体类型，具体应用请参考 14.4 节中的第一个样例工程。

2．其他任务同步与通信方式

除了轻量级事件外，MQX-Lite 还提供了轻量级消息队列、轻量级信号量、互斥量等方式来实现任务间的同步与通信。这些任务间同步与通信方式的详细信息，请参考 MQX-Lite 的用户手册及本书作者编写的《嵌入式实时操作系统 MQX 应用开发技术》一书。

14.2.3 中断处理机制

MQX-Lite 中断服务例程分为内核 ISR（_int_kernel_isr）和用户 ISR 两个相对独立部分。

1．内核 ISR

内核 ISR 与特定处理器相关联，用于实现硬件中断到用户 ISR 的映射，一般通过汇编语

言实现，以确保 MQX-Lite 对中断事件的快速响应。例如，面向 Cortex-M0+处理器，Freescale 给出的用汇编语言编写的内核 ISR 程序函数_int_kernel_isr，在“..\09_MQXLite\psp\core\M0\dispatch.S”文件中。用户 ISR 通常由用户使用 C 语言编写，实现具体功能。

硬件中断向量表（在“..\03_MCU\startup_SKEAZ1284.S”文件中）几乎把所有的中断向量都预设为内核 ISR。复位向量由硬件决定，不受 MQX-Lite 管理。其他的所有外设中断及系统定时器中断（Systick）都是从_int_kernel_isr 这个通用入口来处理的。

2．用户 ISR

用户 ISR 不是任务，而是一个规模小、速度快、能对硬件中断迅速反应的例程，主要功能有重置设备、获取设备数据，以及通过非阻塞式 MQX-Lite 函数向相关任务发送信号等。为了减小堆栈规模，MQX-Lite 支持独立的中断栈，用以保存被嵌套的用户 ISR 的上下文。

用户 ISR 要能够得到正确调用，须在系统初始化时调用_int_install_isr()函数，将用户 ISR 安装到用户中断向量表中。_int_install_isr()函数有三个参数，分别为中断向量号、中断服务例程入口地址（中断函数名称）、中断服务例程使用的参数。

例如，对于串口 2 中断服务例程（UART2_RX_ISR）的 ISR 安装，在 MQX-Lite 自启动任务中包含下列语句。

```
_int_install_isr(UART2_IRQn+16,UART2_RX_ISR,NULL); //安装串口 2 的 ISR
```

其中：

实参“UART2_IRQn+16”：串口 2 的中断向量号。UART2_IRQn 为串口 2 的 IRQ 号，为 14。再加上 16 后得到串口 2 的中断向量号 30。在芯片头文件“..\03_MCU\ SKEA1284.h”中，用 IRQn_Type 定义了一组枚举值代表 IRQ 号，以便直观使用。KEA128 芯片仅提供前 48 个中断的 IRQ 编号。实际编程时，需要用户在这个文件中找到相应中断 IRQ 号的枚举值，加上 16 得到对应的中断向量号，才能安装对应中断。

实参“UART2_RX_ISR”：用户编写的串口 2 中断服务例程的函数名。

实参“NULL”：UART2_RX_ISR 所使用的参数首地址，这里设为“NULL”，即 UART2_RX_ISR 不需要参数。

当中断事件到来时，MQX-Lite 执行如下操作。

（1）进入内核 ISR，执行用汇编语言编写的内核 ISR（_int_kernel_isr）。

（2）使用中断栈。

（3）保存 MSP 到内核数据区。

（4）运行用户 ISR。

（5）用户 ISR 返回后，返回内核 ISR。

（6）返回中断前的任务之后，可有两种情况：第一，若中断期间，未改变任务优先级，则返回中断前的任务，继续执行；第二，若中断期间，使得某一比被中断任务优先级更高的任务就绪，当中断程序执行完毕后，立即重新调度。

14.3 基于 MQX-Lite 的 KEA128 工程框架

本节中基于 MQX-Lite 的 KEA128 工程框架是在 4.4.1 小节中提出的无操作系统工程组织框架的基础之上搭建起来的。与无操作系统下的工程差别仅在于添加了名为“09_MQXLiteLite”的文件夹，此文件夹中存放了 MQX-Lite 内核、PSP（内核支持包）及用户的应用任务代码。

MQXLite 文件夹内包含了 app、bsp、config、include、kernel（内核）、psp 共 6 个下级文件夹，表 14-3 展示了 MQXLite 文件夹下的文件与目录组成。

表 14-3 MQXLite 文件夹的下级文件夹组成

文 件 夹	内含文件说明
app	app_inc.h 任务公共头文件
	task_templates.c任务模板列表文件
	task_main.c自启动任务文件
	task_xxx.c 用户应用任务文件
bsp	开发板相关配置文件，由用户自行编写
config	user_config.h 用于配置文件，配置、裁剪 MQX-Lite 功能
include	存放 MQX-Lite 系统的所有源码头文件
kernel	存放 MQX-Lite 内核功能源代码
psp	与芯片相关的调度、中断等汇编源程序

其中<app>文件夹是按照 MQX-Lite 实时操作系统任务设计的要求增加的应用程序目录，用户开发的应用程序任务主要在此目录下进行操作，该文件夹所包含的文件及其内容见表 14-4。它包含了应用软件的总头文件 app_inc.h、任务模板列表文件 task_templates.c 和自启动任务文件 task_main.c，这三个文件的内容可根据工程要求进行修改，但文件名及文件内容布局不再更改，这是每个工程的共性部分，有利于工程的移植。而“task_xxx.c”为自定义任务，“xxx”代表具体任务名。

表 14-4 app 文件夹的内容

文 件 名	名 称	内 容
app_inc.h	任务公共头文件	用于定义任务模板号、任务的声明，文件名不变，内容可修改
task_main.c	MQX 自启动任务	用于创建用户自定义的任务，文件名不变，内容随着用户任务变动而修改
task_templates.c	任务模板列表	用于向 MQX-Lite 操作系统登记任务，在模板列表中，任务模板编号、任务函数体、任务栈大小、任务优先级、任务启动属性等；文件名不变，内容随着用户任务变动而修改
task_xxx.c	自定义任务	该函数用于实现自定义任务的功能，“xxx”被实际任务名取代

<bsp>文件夹下包含开发板相关的配置及初始化，可参照完整版的 MQX 中提供的 BSP 模板，由用户自行编写。<config>文件夹下存放了 user_config.h 配置文件，用于配置、裁剪 MQX-Lite 的功能。而剩下的<include>、<kernel>及<psp>文件夹下的文件是从 PE 安装目录下的 MQX-Lite 子目录“..\KDS_3.0.0\eclipse\ProcessorExpert\lib\mqxlite\V1.1.1\source”中提取来的。

与无操作系统下的编程相比，原来放在 main.c 函数中的应用程序在 MQX-Lite 中被划分成了各个不同功能的应用任务函数（task_xxx.c），而 main.c 中只需调用_mqxlite_init()及_mqxlite()函数来初始化并启动 MQX-Lite，让操作系统来进行任务的管理与调度。



14.4　KEA128 在 MQX-Lite 下的第一个样例工程

14.4.1　样例工程的功能

KEA128 在 MQX-Lite 下的第一个样例工程以上节给出的框架为模板，完成指示灯的流水闪烁与串口接收发送数据功能，具体要求如下。

硬件连接：

（1）灯 1、灯 2、灯 3、灯 4 硬件连接到 TRK-128 评估板的 D3、D5、D6 及 D8，由 KEA128 芯片的 PTC0、PTC1、PTC2 和 PTC3 引脚控制，高电平灯灭，低电平灯亮。

（2）串口 2 作为缺省通信设备与外界通信，硬件连接于 TRK-128 评估板 KEA128 芯片的 PTD6、PTD7 引脚，作为串口 2 的收、发引脚。

软件功能：

样例工程包含 6 个任务和一个中断服务例程，简要说明如下。

（1）任务 1：自启动任务，初始化外设，创建其他任务。

（2）任务 2：小灯 1（D3）每 4 s 亮暗一次。

（3）任务 3：在小灯 1 之后，小灯 2（D5）每 4 s 亮暗一次。

（4）任务 4：在小灯 1 之后，小灯 3（D6）每 4 s 亮暗一次。

（5）任务 5：在小灯 1 之后，小灯 4（D8）每 4 s 亮暗一次。

（6）任务 6：使用串口 2 作为通信端口，当接收到完整的数据帧时，将整个数据帧内容发送至 PC。

（7）串口 2 中断：接收串口数据组成数据帧。接收数据的格式：“帧头（‘A’）+ 数据长度（1 字节）+ 有效数据 +帧尾（‘D’）”。数据长度是指“有效数据”的字节数。串口 3 初始化波特率为 9600，1 位停止位，无校验。

14.4.2　样例工程任务设计

依据样例工程的功能要求，需在工程“09_MQXLite\app”目录中添加相关的任务函数，包括：小灯闪烁任务函数 task_A()、task_B()、task_C()、task_D()、串口 4 发送数据到 PC 的 task_E()任务函数、自启动任务函数 task_main()、应用任务公共头文件 app_inc.h 及任务模板列表文件 task_templates.c 等。

1．小灯闪烁任务函数 task_A()

小灯 2 任务函数 task_B()和小灯 1、小灯 3～4 的任务函数基本功能和实现方法一样，区别在于任务循环体前的延时时长不同。小灯 1～4 都是每 4 s 亮暗一次，小灯 2～4 在首次亮暗前依次多延时 1 s。小灯使用的引脚可在工程目录“..\06_App_Component\light\light.h”文件中进行配置。下面仅对小灯 2 的程序实现进行分析，这里给出任务函数 task_B()的源码。

```
#include "app_inc.h"

//===========================================================================
//任务名称：task_B
//功能概要：灯 2 每 4 s 亮暗一次，首次亮暗前延时 1 s
//参数说明：未使用
//===========================================================================
void task_B(uint_32 initial_data)
{
    _time_delay_ticks(DELAY_TIMES);
    while (TRUE)
    {
        printf("---Light2 Toggle---\r\n");
        light_change(LIGHT_2);              //改变小灯 2 亮灭状态
        _time_delay_ticks(DELAY_TIMES*4);   //延时 DELAY_TIMES(放弃 CPU 控制权)
    }
}
```

在 while(TRUE)永久循环体内，通过_time_delay_ticks 函数实现时间的延时，DELAY_TIMES 为在 app_inc.h 中定义的宏，其值为 200，即每隔 200×4=800 个时钟滴答执行一次（程序中默认 1 个时钟滴答为 5 ms，可在 bsp.h 中修改，800×5 ms=4 s），切换灯的亮暗，在延时期间操作系统可以调度执行其他的任务。

2．帧数据发送任务函数 task_E()

任务中使用 UART2 作为调试串口 UART_Debug，可在“..\09_MQX\app\app_inc.h”中修改为其他串口模块，而 UART2 使用的收发引脚可在“..\05_Driver\uart\uart.h”中进行配置。对于串口 2 通信的实现包含接收中断的功能，依据 ISR 设计的基本原则，中断服务例程的设计尽可能简短，以便其他异步中断事件能得到响应。在串口 2 接收中断服务例程 UART2_RX_ISR()中只完成数据接收到全局变量数组 g_UART_ISR_buffer[]中，再由 task_E()任务函数将全局变量中的数据通过串口发送到 PC。中断服务例程“UART2_RX_ISR()”位于工程\08_source\isr.c 文件中，这是兼容无操作系统的中断服务例程进行布置的。实现代码如下。

```
#include "app_inc.h"    //应用任务公共头文件

//===========================================================================
//任务名称：task_E
//功能概要：将串口 2 中断接收的完整数据帧发送至 PC
//参数说明：未使用
//备注：使用全局变量 lwevent_group1, UART2_RecData_Event,g_UART_ISR_buffer
//===========================================================================
void task_E (uint32_t initial_data )
{
    int itemp;
    //进入主循环
    while (TRUE)
    {
```

```
        //等待串口 2 接收完整数据帧事件位（Event_UART2_ReData）
        _lwevent_wait_ticks(&lwevent_group1, Event_UART2_ReData, FALSE, 0);
        g_UART_FrameCount++;         //接收的帧数加 1

        printf("串口 2 接收的完整数据帧：(--0x%02X--)",g_UART_FrameCount);

        //发送帧数据
        for(itemp=0;itemp<g_UART_ISR_buffer[1]+3;itemp++)
        {
            printf("%x",g_UART_ISR_buffer[itemp]);
        }
        printf("\r\n");
        //清除串口 2 接收完整数据帧事件位（Event_UART2_ReData）
        _lwevent_clear(&lwevent_group1, Event_UART2_ReData);
    }
}
```

3．任务模板和任务公共头文件

编写好以上各任务后，必须在任务模板列表文件 task_templates.c 中按任务模板的格式进行登记，设置任务栈的大小、任务优先级及任务属性等，为任务创建做好准备。并在任务公共头文件 app_inc.h 中静态分配好任务栈，定义任务模板索引号，对各个任务进行申明。**需要注意的是任务模板列表的最后一个任务模板必须为{0}**，它表示任务模板列表的结束。在任务创建过程中，经常需从任务模板列表的第一条信息，按模板索引号进行查找，若找到的模板索引号为 0，表示已经到模板列表最后一项，查找过程结束。

（1）task_templates.c。

```
#include "app_inc.h"
//为自启动任务注册任务栈
const uint_8 * mqx_task_stack_pointers[] =
{
    task_main_stack,
    NULL
};
//任务模板列表
const TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
    //任务编号,  任务函数,   任务栈大小,        优先级, 任务名,  任务属性
    {TASK_MAIN, task_main, TASK_MAIN_STACK_SIZE, 9,  "main",  MQX_AUTO_START_TASK},
    {TASK_A,    task_A,   TASK_A_STACK_SIZE,    10,  "task_A",  0},
    {TASK_B,    task_B,   TASK_B_STACK_SIZE,    10,  "task_B",  0},
    {TASK_C,    task_C,   TASK_C_STACK_SIZE,    10,  "task_C",  0},
    {TASK_D,    task_D,   TASK_D_STACK_SIZE,    10,  "task_D",  0},
    {TASK_E,    task_E,   TASK_E_STACK_SIZE,     8,  "task_E",   0},
    {0}
};
```

（2）app_inc.h。

```
#ifndef __APP_INC_H_
#define __APP_INC_H_
//-----------------------------------------------------------------------

//1.包含头文件
#include "isr.h"
#include "bsp.h"
#include "mqxlite.h"
#include "mqx_inc.h"
#include "light.h"
#include "lwevent.h"
#include "printf.h"

//2.声明全局变量
//防止全局变量重复声明的前缀处理方法
#ifdef GLOBLE_VAR              //GLOBLE_VAR 在 task_main.c 文件中宏定义
#define G_VAR_PREFIX           //task_main.c 文件中使用全局变量不加“extern”前缀
#else
#define G_VAR_PREFIX  extern   //其他文件中使用全局变量自动加“extern”前缀
#endif
//声明全局变量（全局变量类型前一律前缀 G_VAR_PREFIX）
G_VAR_PREFIX uint8_t g_UART_ISR_buffer[100];  //存放 UART2 接收的数据帧
G_VAR_PREFIX uint8_t g_UART_FrameCount;       //UART2 接收的数据帧的个数
G_VAR_PREFIX LWEVENT_STRUCT  lwevent_group1;  //轻量级事件组
   #define  Event_UART2_ReData   ((1uL)<<(6)) //串口 2 接收完整数据帧事件位
#define DELAY_TIMES    200                    //每个 tick 对应 5ms

//3.登记任务模板编号
#define TASK_MAIN                  1
#define TASK_A                     2
#define TASK_B                     3
#define TASK_C                     4
#define TASK_D                     5
#define TASK_E                     6

//4.为任务创建任务栈
#define TASK_MAIN_STACK_SIZE (sizeof(TD_STRUCT)+1024+PSP_STACK_ALIGNMENT+1)
#define TASK_A_STACK_SIZE (sizeof(TD_STRUCT) + 512 + PSP_STACK_ALIGNMENT+1)
#define TASK_B_STACK_SIZE (sizeof(TD_STRUCT) + 512 + PSP_STACK_ALIGNMENT+1)
#define TASK_C_STACK_SIZE (sizeof(TD_STRUCT) + 512 + PSP_STACK_ALIGNMENT+1)
#define TASK_D_STACK_SIZE (sizeof(TD_STRUCT) + 512 + PSP_STACK_ALIGNMENT+1)
#define TASK_E_STACK_SIZE (sizeof(TD_STRUCT) + 512 + PSP_STACK_ALIGNMENT+1)

G_VAR_PREFIX uint_8 task_main_stack[TASK_MAIN_STACK_SIZE];
G_VAR_PREFIX uint_8 task_A_stack[TASK_A_STACK_SIZE];
G_VAR_PREFIX uint_8 task_B_stack[TASK_B_STACK_SIZE];
```

```
G_VAR_PREFIX uint_8 task_C_stack[TASK_C_STACK_SIZE];
G_VAR_PREFIX uint_8 task_D_stack[TASK_D_STACK_SIZE];
G_VAR_PREFIX uint_8 task_E_stack[TASK_D_STACK_SIZE];

//5.声明任务函数
void task_main(uint32_t initial_data);
void task_A(uint32_t initial_data);
void task_B(uint32_t initial_data);
void task_C(uint32_t initial_data);
void task_D(uint32_t initial_data);
void task_E(uint32_t initial_data);

//6.声明中断服务例程 ISR
void UART2_RX_ISR(pointer user_isr_ptr);

//--------------------------------------------------------------------
#endif    //app_inc.h
```

4．自启动任务及任务的创建

自启动任务 task_main 是 MQX-Lite 启动后执行的第一个应用任务，它是在 MQX-Lite 系统初始化时由_mqxlite()函数创建的。自启动任务的功能就是调用_task_create_at()函数创建其他的任务。根据给定的任务索引号，_task_create_at()函数从模板列表查找任务的相关信息创建任务，即把任务的相关信息填写到任务描述符节点，并把该节点加入到任务就绪队列，供任务调度器调度。

```
#define GLOBLE_VAR

#include "app_inc.h"

//====================================================================
//任务名称：task_main
//功能概要：MQX 自启动任务，主要实现全局变量初始化、创建其他任务、安装用户 ISR
//参数说明：未使用
//====================================================================

void task_main(uint32_t initial_data)
{
    //1.全局变量初始化
    _lwevent_create(&lwevent_group1,0);         //创建轻量级事件组
    g_UART_FrameCount=0;                        //接收的帧数

    //2. 关总中断
    DISABLE_INTERRUPTS;

    //3.外设初始化
    light_init(LIGHT_1,LIGHT_OFF);              //小灯 1 初始化
```

```
    light_init(LIGHT_2,LIGHT_OFF);              //小灯 2 初始化
    light_init(LIGHT_3,LIGHT_OFF);              //小灯 3 初始化
    light_init(LIGHT_4,LIGHT_OFF);              //小灯 4 初始化
    uart_init(UART_Debug,9600);                 //串口 2 使用总线时钟 24 MHz
    printf("Hello Uart_%d!\r\n",UART_Debug);    //串口发送初始化提示

    //4.创建其他任务
    _task_create_at(0, TASK_A, 0, task_A_stack, TASK_A_STACK_SIZE);
    _task_create_at(0, TASK_B, 0, task_B_stack, TASK_B_STACK_SIZE);
    _task_create_at(0, TASK_C, 0, task_C_stack, TASK_C_STACK_SIZE);
    _task_create_at(0, TASK_D, 0, task_D_stack, TASK_D_STACK_SIZE);
    _task_create_at(0, TASK_E, 0, task_E_stack, TASK_E_STACK_SIZE);

    //5.安装用户 ISR
    //注册调试串口的 ISR
    _int_install_isr(UART0_IRQn+UART_Debug+16,UART2_RX_ISR,NULL);

    //6.使能模块中断及总中断
    uart_enable_re_int(UART_Debug);         //使能调试串口接收中断

    //7.开总中断
    ENABLE_INTERRUPTS;

    //-----------------执行完毕，本任务进入阻塞态-----------------------------
    _task_block();
}
```

5. 串口 2 中断服务例程

串口 2 中断服务例程从串口接收数据组建数据帧，并回显接收到的数据。定义的串口数据帧格式为：帧头（1B）+数据长度（1B）+数据内容（NB）+帧尾（1B），此处使用字符“A”（0x41）作为帧头，字符“D”（0x44）作为帧尾。当接收到一个完整的数据帧时，置接收成功事件位，唤醒帧数据发送任务函数 task_E()。若最后一个帧尾数据不为“D”，则认为前面收到的数据均为无效数据并重新开始接收计数。

```
//============================================================================
//文件名称：isr.c
//功能概要：中断底层驱动构件源文件
//版权所有：苏州大学飞思卡尔嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2015-4-26  V1.0
//芯片类型：KEA128
//============================================================================

#include "includes.h"
#include "app_inc.h"

#define FrameHead    (0x41)        //帧头
#define FrameTail    (0x44)        //帧尾
```

```
//内部函数声明
uint8_t CreateFrame(uint8_t Data,uint8_t * buffer);

////=========================中断函数服务例程==============================

//=========================================================================
//中断例程：UART2_RX_ISR
//功能概要：串口 2 中断服务例程，每收到一个字节执行一次本程序，发送回去，并组帧
//参数说明：user_isr_ptr（未使用）
//备注：使用全局变量 g_UART_ISR_buffer[]
//=========================================================================
void UART2_RX_ISR( pointer user_isr_ptr)
{
    uint8_t Redata;                          //存放接收来的数据
    uint_8 flag = 1;

    Redata = uart_re1(UART_2, &flag);        //接收的一个字节数据
    if (flag!=0) goto UART2_RX_ISR_End;
    else uart_send1(UART_2, Redata);         //将收到的一个字节数据发送回去
    //组帧成功,置“串口 2 接收完整数据帧事件位”
    if(CreateFrame(Redata,g_UART_ISR_buffer)!=0)
    {
        _lwevent_set(&lwevent_group1,Event_UART2_ReData);
    }

UART2_RX_ISR_End: ;
}

//内部调用函数

//=========================================================================
//ISR 名称：Createbuffer
//功能概要：组建数据帧，将待组帧数据加入到数据帧中
//参数说明：Data:待组帧数据
//          buffer:数据帧变量
//函数返回：组帧状态，0-组帧未成功；1-组帧成功
//备注：十六进制数据帧格式
//          帧头       +   数据长   + 有效数据     +   帧尾
//          FrameHead  + len     + 有效数据     +   FrameTail
//=========================================================================
uint8_t CreateFrame(uint8_t Data,uint8_t * buffer)
{
    static uint8_t frameCount=0;             //组帧计数器
    uint8_t frameFlag;                       //组帧状态
    frameFlag=0;
    //根据静态变量 frameCount 组帧
    switch(frameCount)
```

```
    {
        case 0:                                  //第一个数据
        {
            if (Data==FrameHead)                 //收到数据是帧头 FrameHead
            {
                buffer[0]=Data;
                frameCount++;
                frameFlag=0;                     //组帧开始
            }
            break;
        }
        case 1:                                  //第二个数据，该数据是随后接收的数据个数
        {
            buffer[1]=Data;
            frameCount++;
            break;
        }
        default:                                 //其他情况
        {
            //第二位数据是有效数据长度,根据它接收余下的数据直到帧尾前一位
            if(frameCount>=2 && frameCount<=(buffer[1] + 1))
            {
                buffer[frameCount]=Data;
                frameCount++;
                break;
            }

            //若是末尾数据则执行
            if(frameCount>=(buffer[1]+2))
            {
                if (Data==FrameTail)             //若是帧尾
                {
                    buffer[frameCount]=Data;     //将帧尾 0x44 存入缓冲区
                    frameFlag=1;                 //组帧成功
                }
                frameCount=0;                    //计数清 0，准备重新组帧
                break;
            }
        }
    }                                            //switch_END
    return frameFlag;                            //返回组帧状态
}
```

14.4.3 样例工程的执行流程及运行结果

样例工程中的任务信息如表 14-5 所示。样例工程在启动后进入 main()函数，在其中调用 _mqxlite_init 函数及_mqxlite 函数来初始化并启动 MQX-Lite 操作系统，在进行一系列的初始化过程之后，进入自启动任务 task_main。

表 14-5　样例任务

任 务 函 数	优　先　级	任 务 属 性	任 务 功 能
task_main	9	自启动	打印输出唤醒消息，启动其他任务
task_A～task_D	10	无	以 4 s 为周期闪烁小灯 1～4
task_E	8	无	通过串口 4 接收 PC 数据，回发相应数据的功能

在自启动任务 task_main 中，初始化所用到的外部设备，并依次创建其他任务 task_A、task_B、task_C、task_D 及 task_E，最后调用 task_block()函数将自身阻塞。此后 MQX 将按照所创建任务的优先级对任务进行调度。

需要注意的是，在创建其他任务的过程中，任务 task_A 到 task_D 的优先级（都为 10）低于自启动任务 task_main 的优先级（9），所以在任务创建后这两个任务会进入各自优先级的任务就绪队列中等待调度；而在创建任务 task_E 后，由于其任务优先级（8）高于自启动任务 task_main（9），所以它会在下一个系统调度时刻（1 个 systick 的时间）到来时抢占当前任务开始执行。而由于在 task_E 的任务函数循环体开始时使用了轻量级事件来同步任务，在串口 2 没有接收到完整的数据帧时，task_E 被轻量级事件阻塞，系统会重新回到 task_main 任务继续执行。

将样例工程编译下载到目标板，重新上电后，可以观察小灯 1～4 按设定的方案不停闪烁。通过串口 2 与 PC 连接，在串口调试器中向目标板发送数据，可看到其回显到数据接收框中。从串口调试器以十六进制发送“410331323344”一帧数据，在显示窗口回送了数据帧的内容，如图 14-1 所示。



图 14-1　样例工程运行结果

第 15 章

基于 KEA 的无刷直流电机的汽车应用

本章导读：本章主要介绍基于 KEA 的无刷直流电机汽车应用。首先介绍无刷直流电机（简称 BLDC）在汽车领域的应用现状和发展趋势；接着就开始讲 BLDC 操作指南，如何一步一步地将一个 BLDC 在 KEA128-BLDCRD 板子上运转起来的例子，然后剖析 BLDC 的工作原理，以及无传感器控制算法；最后给出应用实例——车用冷却风扇，并对实例的方案、硬件结构、电气指标等做进一步介绍。

15.1 无刷直流电机在汽车上的应用现状和发展趋势

随着汽车工业的不断发展和电子技术的不断革新，电机在汽车上的应用越来越广泛。各种电机的控制让汽车变得更加智能和更加人性化。无论是在助力转向系统，还是在车窗控制，座椅调节及加热，又或是空调控制，引擎冷却风扇和油泵水泵等，到处都可以看到电机在发挥着作用。根据某咨询机构的预计，到 2016 年，平均每台车上将会有近 40 个电机。

车用电机的发展趋势中一个明显的特点就是无刷化。传统的有刷电机由于其存在换向器磨损及换向时存在火花等问题，正在被无刷直流电机所取代。

车用电机另一个发展趋势是从以前的有传感器控制转向无传感器控制，这就要求一些复杂的算法和强大的微处理器来支持。

飞思卡尔的 KEA 系列产品就非常适合无刷直流电机的控制。本章会重点讲述无刷直流电机的无传感器控制的基本知识和控制策略，以及开发过程中的注意事项等，最后介绍一个车用冷却风扇的实例。

15.2 无刷直流电机在 KEA128-BLDCRD 板上的操作指南

其实运转一个 BLDC（Brushless Direct Current Motor，无刷直流电机）并不像想象得那么复杂，即便无传感器控制，也非常轻松。接下来就让我们使用无传感器算法在 KEA128-BLDCRD 板子上跑一个电机。

15.2.1 软硬件的准备

1．硬件

（1）一块 KEA128-BLDCRD 板子，可以直接在飞思卡尔的官网中输入“KEA128-BLDCRD”进行在线订购。

（2）一个 BLDC 电机，推荐 Linix 的 45ZWN24-90-B，参数为 24 V（可以 12 V 运行），额定转速为 9350RPM，额定功率为 90 W。

（3）12 V 直流带限流保护稳压电源。

2．软件

（1）在飞思卡尔官网 freescale.com/KEA128-BLDCRD 或者本书网上光盘的 15 章节中下载和安装 KEA 无传感器 BLDC 驱动程序包。

（2）在飞思卡尔官网 freescale.com/FREEMASTER 中下载和安装最新版本的 FreeMASTER。

15.2.2 操作流程

（1）将电机的三根相线连接到开发板的 J8、J9 和 J10。

（2）将 12 V 的稳压电源的两端连接到开发板的 J11（接 12 V）和 J12（接 GND），稳压电源的电压范围是 8～18 V，注意该电压值会影响电机的最大速度。

（3）通过一个 USB 线连接个人电脑和开发板，允许电脑自动配置 USB 驱动。

（4）打开 KDS3.0，选择 File-Import，选择 General-Existing Projects into Workspace，选择 Select root directory 然后单击 Browse 按钮，找到网上光盘“..\CH15- KEA128-BLDC-Sensorless-KDS”然后单击确认按钮；再单击完成按钮，然后选择 RUN 目录下的 Debug，选择 KEA128_FLASH_OpenSDA 调试工具进行调试和烧录。

（5）启动 FreeMASTER 应用程序，通过选择 File 中的 Open Project 打开网上光盘“..\CH15-KEA128-BLDC-Sensorless-KDS\KEA128_BLDC_Sensorless.pmp”文件；通过选择菜单 Project 中的 Options…设定 RS232 波特率为 115200，同时设定好 COM 口；单击 FreeMASTER 工具栏中的红色的 STOP 按钮或者使用快捷键 Ctrl+K 来使能 RS232 通信。如果设定成功的话，状态栏中会显示“RS232;COMn;speed=115200”。

（6）单击 FreeMASTER 内嵌的电机控制应用调试工具（MCAT）中的 App Control 菜单，会出现主调试界面。

（7）控制电机的方式如下。

第一，电机的运行方向可通过开发板上的开关 SW3 来设定。

第二，启动电机有两种方法：一种是通过开发板上的开关 SW1；另一种是按下 MCAT 主调试界面下的 ON/OFF 虚拟开关。

第三，改变电机转速有两种方法：一种是通过按开发板上的开关 SW1 来加速，按开关 SW2 来减速；另一种来是通过双击 MCAT 主界面下的虚拟速度表或者直接修改观测窗口“requiredSpeed”的变量值来改变转速。

第四，停止电机也有两种办法：一种是通过同时按下开关 SW1 和 SW2；另一种是通过按下 MCAT 主调试界面下的 ON/OFF 虚拟开关。

第五，一旦有故障发生，可以同时按下开发板上的 SW1 和 SW2 来清除故障或者在 MCAT 主调试界面下单击绿色的 Clear Faults 按钮来清除。

看到 BLDC 运转起来后是不是很有趣，但 BLDC 会什么会转呢？而神奇的无传感器技术又究竟是怎么回事呢？接下来让我们从 BLDC 的原理讲起。

15.3　无刷直流电机驱动的基本原理及无传感器控制

无刷直流电机（BLDC）相当于把有刷直流电机的定转子进行交换，同时将传统的机械换向变成了用功率器件实现的电子换向。一个典型的三相 BLDC 剖面示意图如图 15-1 所示。

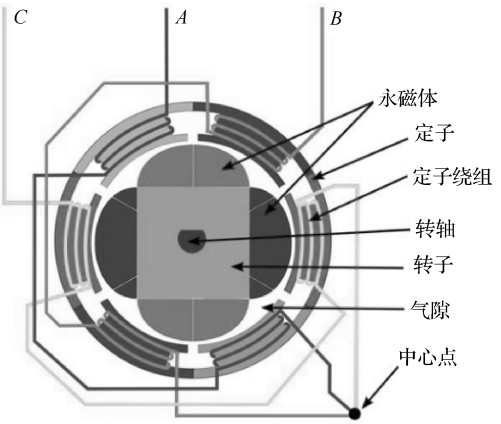


图 15-1　BLDC 剖面示意图

由图 15-1 可以看到该 BLDC 转子上贴了 4 片磁钢，相当于两对极。定子三相绕组采用集中绕组形式，并内部星形连接。

BLDC 运转起来后，将会产生梯形波的反电动势波形，如果忽略其高次谐波部分，其反电动势（e_a,e_b,e_c）如图 15-2 所示，可以看出每半个周期都有 120° 电角度的固定幅值段和 60° 电角度的过渡段。理想的三相电流（i_a,i_b,i_c）每半个周期也需要 120° 导通角且电流波形是方波输出。为了获得较平稳的力矩输出，每相的相电流必须和反电动势同步，也就是说，相电流的 120° 导通的方波阶段必须和反电动势的 120° 的梯形波的平台部分在相位上重叠。

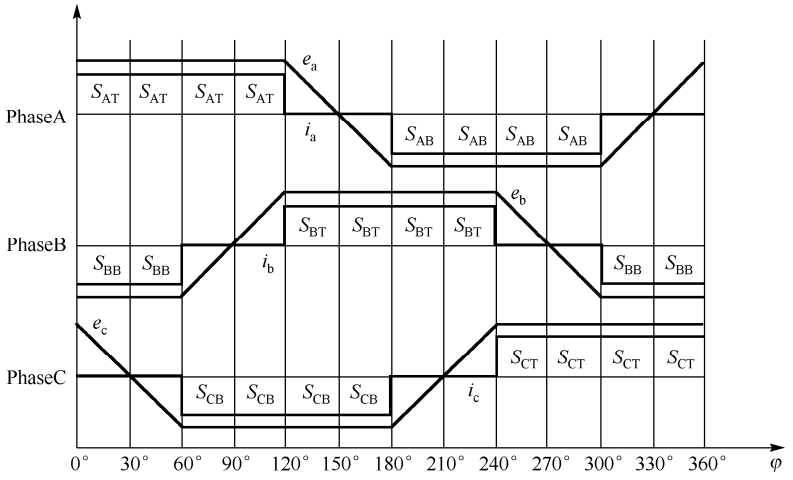


图 15-2　BLDC 反电动势及相电流波形示意图

为了实现同步，就必须要知道反电动势的相位，也就是转子的位置信息，这是 BLDC 控制的关键所在。目前流行的位置传感器主要有霍尔、光电编码器及旋转变压器等。另外，需要提到一点的是无传感器技术在风机、水泵和压缩机等流体类控制系统里使用的越来越多，主要有以下几类无传感器控制策略。

（1）反电动势过零检测。

（2）磁通估计。

（3）各种系统状态观测器。

（4）高频注入。

以上无传感器控制策略各有优缺点，用户可以根据自己的需要做进一步的了解。从 BLDC 控制的角度来讲，最重要的就是换向控制和转速转矩控制。

15.3.1 换向控制

三相 BLDC 的功率部分一般采样三相桥驱动 6 个 MOSFET 或者 IGBT，如图 15-3 所示。

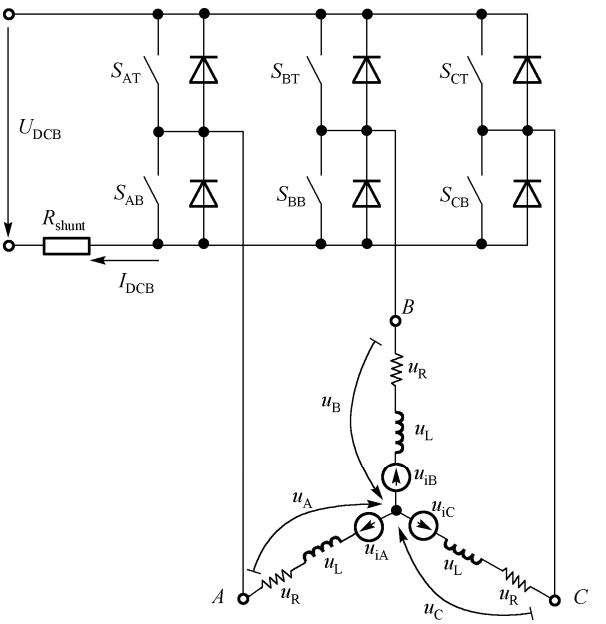


图 15-3 三相 BLDC 的功率部分拓扑图

对于 BLDC 的换向控制，六步换向方法使用得最为广泛，因此就以它为例来描述下换向策略的实现。在六步换向控制中，BLDC 工作在两相模式下，也就是说，任意一个时刻三相中都只有两相是导通的，而另一相是断开的。具体时刻导通哪两相是由转子的位置决定的，表 15-1 给出了转子位置和 6 个开关导通的逻辑关系。检测到转子位置后，根据这个位置去查表，然后闭合相应的功率管，电机就会转动起来了。

表 15-1　转子位置及功率器件导通逻辑表

转子位置	序　号	开关导通		相电流		
				相位 A	相位 B	相位 C
0°～60°	0	S_{AT}	S_{BB}	+	–	Off
60°～120°	1	S_{AT}	S_{CB}	+	Off	–
120°～180°	2	S_{BT}	S_{CB}	Off	+	–
180°～240°	3	S_{BT}	S_{AB}	–	+	Off
240°～300°	4	S_{CT}	S_{AB}	–	Off	+
300°～360°	5	S_{CT}	S_{BB}	Off	–	+

15.3.2　转速转矩控制

换向控制是确保 BLDC 的相电流在一个正确的相位上，而转速转矩控制则是控制相电流的幅值，使其满足系统对转速转矩的要求。那如何实现对相电流幅值的连续控制呢？PWM 控制就可以做到。

PWM 控制在各种微处理器应用中被广泛使用。在 BLDC 控制的应用里，PWM 模块的占空比 Duty 就是速度 PI 调节器的输出。速度 PI 调节器放大目标速度和实际速度的误差，然后对放大值进行适当的缩放，使其满足 PWM 占空比的范围。

BLDC 的机械速度可以用转子的机械角对时间的微分来计算：

$$\omega_{\text{mech}}=\frac{\mathrm{d}\varphi_{\text{mech}}}{\mathrm{d}t}=\frac{1}{p}\frac{\mathrm{d}\varphi_{\text{e1}}}{\mathrm{d}t}\approx\frac{1}{p}\frac{\Delta\varphi_{\text{e1}}}{\Delta T}$$

因为一个电周期里需要换向 6 次，也就是说一个换向周期对应的就是电周期的 1/6，也就是 60° 的电角度，所以上面的公式可以这么写：

$$\omega_{\text{mech}}=\frac{1}{p}\frac{\Delta\varphi_{\text{e1}}}{\Delta T}=\frac{1}{p}\frac{\frac{360^\circ}{6}}{T_{\text{CM}}}$$

$$=\frac{1}{p}\frac{360^\circ}{T_{(0^\circ\to60^\circ)}+T_{(60^\circ\to120^\circ)}+T_{(120^\circ\to180^\circ)}+T_{(180^\circ\to240^\circ)}+T_{(240^\circ\to300^\circ)}+T_{(300^\circ\to360^\circ)}}=\frac{360^\circ}{p\sum_{n=0}^{5}T_{\text{CM}}^{n}}$$

式中，p 为极对数；T_{CM} 为两次连续换向事件的间隔，也就是换向时间；T_{CM}^{n} 为 Sector 为 n(n=0,1,2,3,4,5)的换向时间；φ_{e1} 为电气位置角度；φ_{mech} 为对应的机械位置角。

15.3.3　互补型极性 PWM 调制技术

前面有讲到，各开关管的状态是由换向控制来决定的，而转矩则是由 PWM 的占空比来决定的。BLDC 采用互补型单极性 PWM 调制技术，可以降低 MOSFET 的开关损耗，同时改进了系统的 EMC 性能。

单极性 PWM 控制意味着电机相上看过去只有正向电压，为了实现单极性 PWM，其实一相工作在互补 PWM 模式，另一相接地，最后一相不通电，如图 15-4 所示。可以看到，以 60° 电角度划分的 6 个区间里，PWM 型式都是一样的，只是相序不同而已。而相序的选择依然是根据转子所在的电角度所在的区间，通过换向事件来实现的。接下来我们就重点来看看如何找到转子的位置。

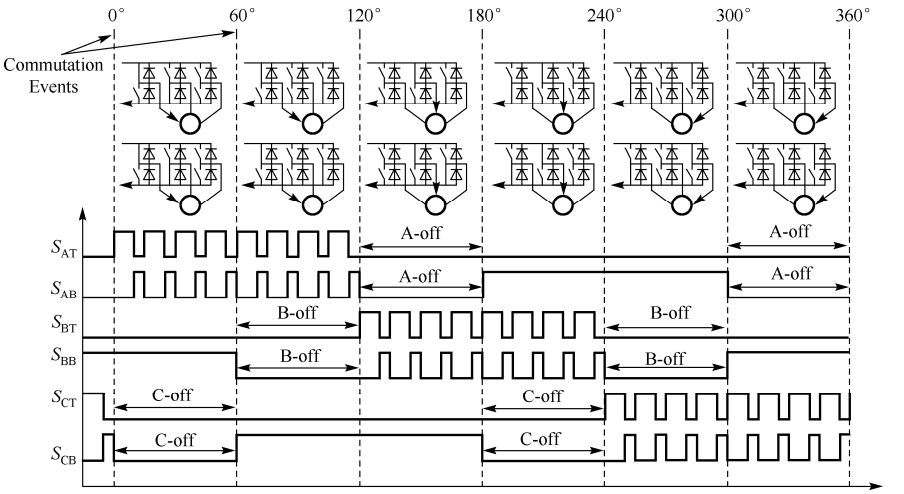


图 15-4　互补型单极性 PWM 控制图

15.3.4　基于反电动过零的位置估计

由图 15-5 可以发现，换向事件都是发生在反电动势过零后 30° 电角度的时刻，而反电动势过零也发生在最近的换向事件后的 30° 电角度时刻。设想一个 BLDC 以一个固定速度运行，那么从最近的换向事件到反电动势过零所需的时间和从反电动势过零到接下来的一个换向事件所需的时间是一样的。在时域中，一个反电动势过零就发生在两次换向事件的中间时刻。因此，反电动势过零事件结合一个定时器，就可以用来估计确切的换向点和计算电机的速度。

1．反电动势过零原理

为了说明反电动势过零的原理，首先可以得到 BLDC 的等效的基本电路拓扑，如图 15-5 所示，接下来根据这个图可以推导出一个简化的数学模型。该模型可以用来找出实际测量的电机波形与反电动势过零之间的内在联系。找到反电动过零，也就可以确定换向事件。

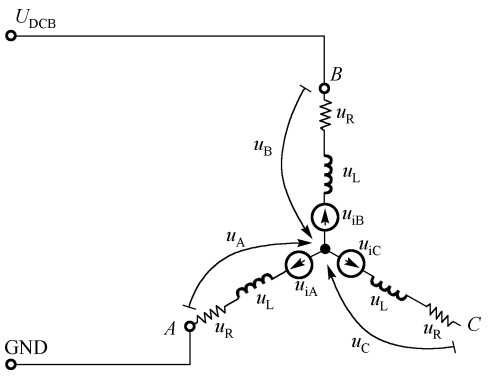


图 15-5　无刷直流电机 A 相 B 相导通时的电路图

该数学模型是基于两相模式，也就说两相导通，一相断开的方式。为了简化该模型的表达式，可以设定模型中性点电压是直流母线电压的一半；同时，该数学模型是假设电机三相是完全对称的。

$$u_N = U_{DCB} - Ri_b - L\frac{di_b}{dt} - e_b$$

$$u_N = Ri_a + L\frac{di_a}{dt} - e_b$$

$$i_a = i_b$$

由上面三个式子可以推导出

$$u_N = \frac{U_{DCB}}{2} - \frac{e_b + e_a}{2}$$

由于三相是对称的，所以所有反电动势之和是 0，因此有 $e_a+e_b+e_c=0$，也就是 $e_c=-(e_b+e_a)$。

没有导通的那相因为没有电流流过，所以其电压方程为

$$u_N=u_C-e_c$$

通过上面的方程联立方程组，不导通的相其相电压为

$$u_C=\frac{U_{DCB}}{2}+\frac{3}{2}e_c$$

综上所述，反电动势过零时刻，顾名思义其反电动势电压是 0（这个例子里 $e_c=0$），可以通过检测不导通相的相电压（u_C）然后和母线电压的一半（$\frac{U_{DCB}}{2}$）进行比较，这样就可以非常准确地找到反电动势过零时刻。

2．反电动势过零事件检测和相电流检测

转子的位置可以通过检测不导通那相的反电动势来实现。如图 15-6 所示，深色窗口代表该相不导通，在这个窗口里检测的电压就是反电动势电压。在反电动势过零事件时刻，用深色窗口中用黑圈标出，刚好发生在换向周期的中点。这个时刻，相电压是等于直流母线电压的一半，就如前文所描述的那样。如果速度是稳定的，两个连续的过零事件的周期就等于换向周期。

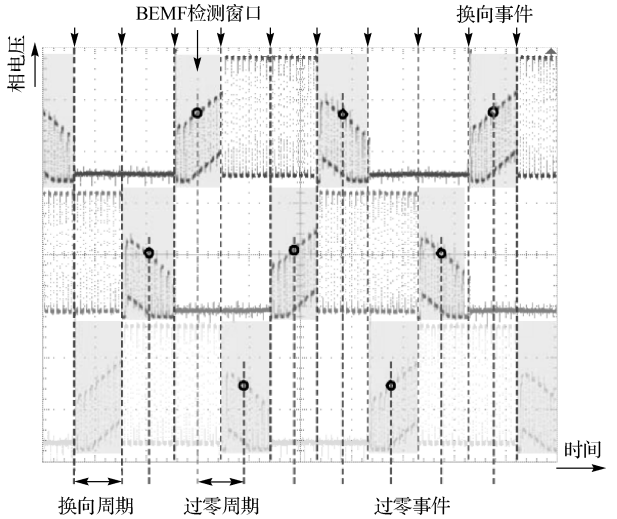


图 15-6　BEMF 过零事件和换向事件以及和单极性互补 PWM 之间的关系

图 15-7 是一个 PWM 周期放大后的状态。图的上部是 PWM 的波形，A 相是由 PWM 控制的，而 C 相是整个 PWM 周期都导通到地的。在 PWM 打开的时候，A 相的上管导通，C 相的下管导通，电流从直流母线流进 A 相，然后通过 C 相和直流母线采样电阻流出。在这个环节里，电机的中性点的电压是 $U_{DCB}/2$。不导通相的反电动势电压会从小于 $U_{DCB}/2$ 值一直变化到大于 $U_{DCB}/2$ 的值，这中间会经历 $U_{DCB}/2$ 时刻，也就是说过零事件是可以被检测到的。同样，采样电阻上的相电流也可以被检测到。在 PWM 周期关断时刻，A 相和 C 相下管都导通，相电流通过 A 相，C 相以及两个下管在内部续流。在这个 PWM 关断的阶段，直流母线采样电阻上是采不到相电流的，与此同时电机的中性点是连到地的，所以反电动势过零也是检测不到的。

综上所述，相电流和反电动势电压采集必须在 PWM 有效（ON）的阶段实现。

3．反电动势检测

通过前面的讨论，知道反电动势检测必须在 PWM 有效（ON）的阶段来实现。需要特别注意的是，因为开关噪声的缘故，这个采样点需要设在 PWM 有效（ON）阶段结束前的时刻，如图 15-8 中标示深色区域阶段。

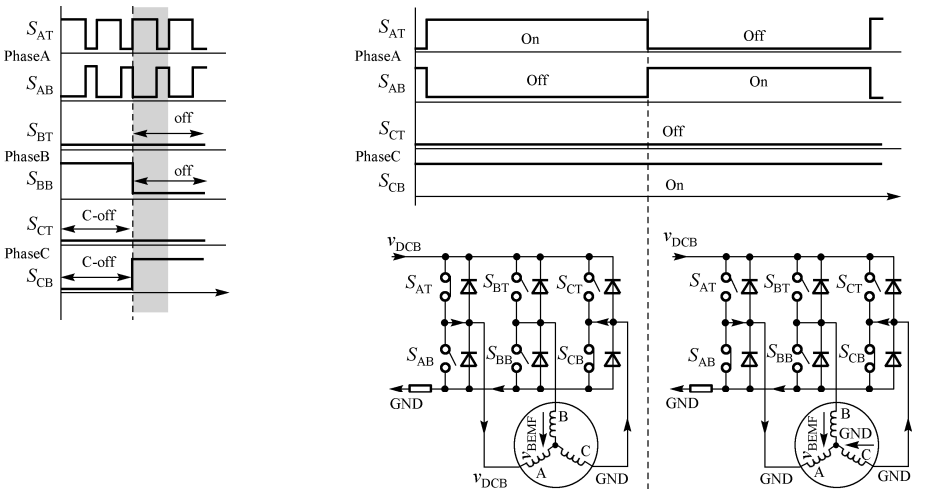


图 15-7　单极性互补 PWM 时的反电动势过零检测

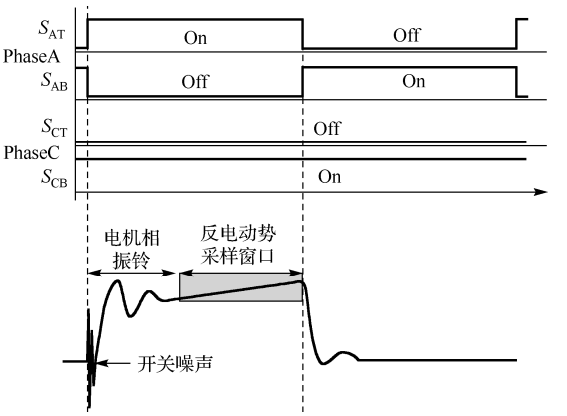


图 15-8　反电动势电压检测

需要注意的是，由于电机和功率级参数的不同，电压振铃的幅度、周期和阻尼也会发生变化，因此建议反电动势电压的采样要靠近采样窗口的尾部。采样点的时刻也需要被记录下来，可以用来增强反电动势过零检测的精准度。

如果放大反电动势电压检测窗口，可以看到过零事件可以发生在任两个连续反电动势电压采样之间，如图 15-9 所示。为了得到更精准的位置估计，一个精确的过零点是必需的，这个精确的过零点是基于两次连续的反电动势电压采样的插值来近似得到的。假设电机没有正在加速，实际的反电动势电压采样在 T_{ADC} 时刻采到电压 e_T，在 $T_{ADC}-T_{PWM}$ 时刻采到电压 e_{T-1}，然后就可以根据下面的方程来计算精确的过零事件。

$$\frac{e_{\mathrm{T}}-e_{T-1}}{T_{\mathrm{PWM}}}=\frac{e_{\mathrm{T}}-\dfrac{U_{\mathrm{DCB}}}{2}}{T_{\mathrm{ADC}}-T_{\mathrm{ZC}}}\geqslant T_{\mathrm{ZC}}=T_{\mathrm{ADC}}-\frac{e_{\mathrm{T}}-\dfrac{U_{\mathrm{DCB}}}{2}}{e_{\mathrm{T}}-e_{T-1}}\cdot T_{\mathrm{PWM}}$$

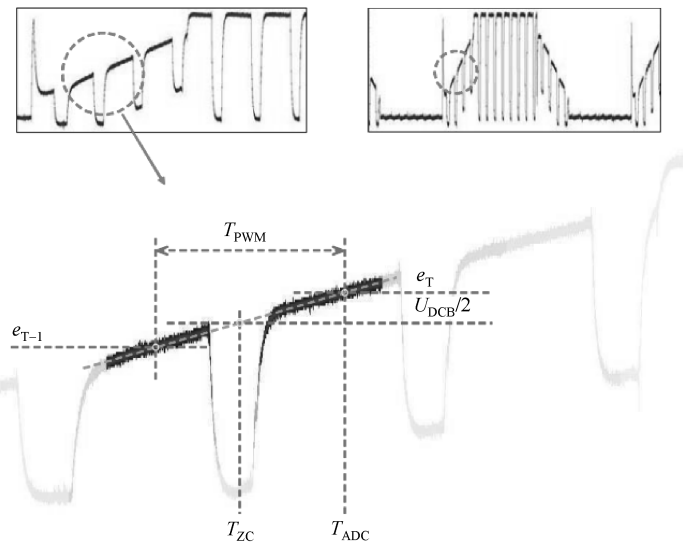


图 15-9　反电动势过零的精确计算

当两次连续的反电动势电压和直流母线电压的一半进行比较，其结果发生变化的时候，可以用这个方程来计算换向周期。为了进一步增强检测过零点事件的准确性，需要保证同时对母线电压和反电动势电压进行采样。

关于反电动势检测，最后需要提到的一点是关于检测过程中的一些限制。当发生换向事件的时刻，某一相刚刚断开，但电流不会马上突变为零，这个时候还是会有电流流过续流二极管的。续流二极管会把刚断开的这一相连到直流母线的正极性端或者负极性端，如图 15-10 所示，续流的时间主要取决于电机的负载和电感。

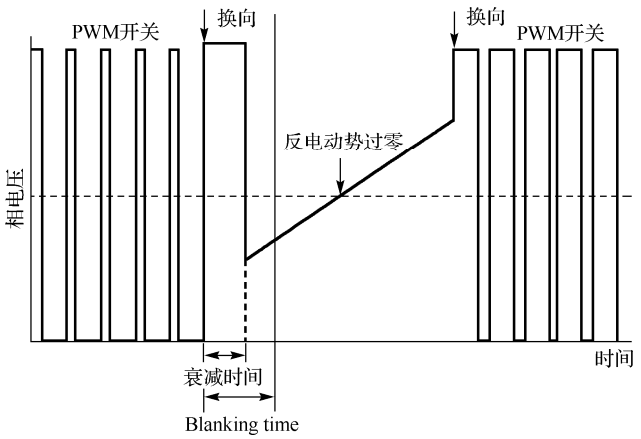


图 15-10　续流现象和 Blanking time

事实上从电压采样角度来看，很难去区分反电动势电压和续流引起的突变电压。为了解决这个问题，在换向时刻发生后引入一个 Blanking time 阶段。在这个阶段里，反电动势电压

不采样，或者采样但不用于过零判断。这个 Blanking time 大小的设定和电机参数、负载大小及动态应用参数都有关系。

15.3.5 无传感器 BLDC 的启动

无传感器 BLDC 的启动一直都是个难点，因为前面讲到过无传感器的实质是检测反电动势过零来判断换向周期，但反电动势大小是随转速变化的，转速低的时候反电动势很小，因此判断其过零就非常困难。也就是说，在低速下，基于反电动势理论的无传感器策略很难找准转子的位置，这样就需要采样传统的三段式启动，即对齐（Alignment）、开环加速（Startup）和闭环运行（Run）。

1．对齐

BLDC 在没有运行之前，是不知道其转子位置的，所以对齐的目标是将转子旋转到一个确定的位置上。有了这个确定的位置就可以保证启动的时候能在正确的方向上能产生最大的力矩。在对齐的过程中，为了能在两个旋转方向上都获得最好的性能，三相都需要导通，其中 C 相连到正母线电压，A 相和 B 相接地。对齐时间主要取决于电机的机械时间常数、负载和所施加的对齐电流。

2．开环加速

在这个阶段，没有任何的转子位置反馈，电机换向控制工作在开环模式下。换向周期是由开环启动曲线来决定的。开环加速阶段开始后，会一直将转子加速到一个较高的转速（通常接近额定转速的 5%）。在这个转速下可以产生可靠的反电动势电压，从而可以确保由开环加速切换到由反电动势过零检测所控制的闭环运行过程中。

3．闭环运行

当反电动过零检测可靠后，系统将由开环切换到闭环运行。闭环运行的框图如图 15-11 所示。反电动势的过零检测用来控制换向事件，电机的转速计算也是基于反电动势过零事件的周期值。目标速度和估计速度的误差送到速度 PI 调节器，其输出的是 PWM 模块的占空比，最后转换为 BLDC 的电压。系统不停地检测电机电流，将其滤波后输入到电流调节器。用电流 PI 调节器的输出来限制速度 PI 调节器，从而保护电机使其免于超过最大允许电流。

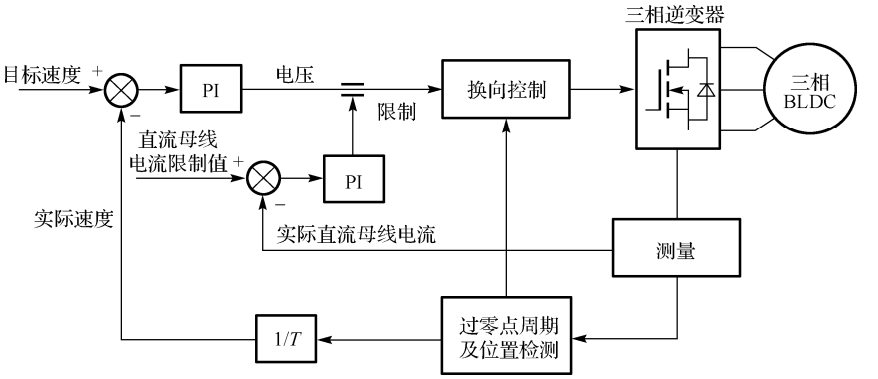


图 15-11 速度控制和电流限制

15.4　基于 KEA 的车用无刷直流电机应用实例——车用冷却风扇

现选取一个车用冷却风扇的实例来看一下 KEA 是如何在车用无刷直流电机控制上发挥其强大功能的。飞思卡尔开发的车用冷却风扇系统总体照片如图 15-12 所示，该演示系统还可以通过 USB 连接到笔记本，使用上位机软件工具来控制冷却风扇启停、转速，以及监控冷却风扇的运行状态等。



图 15-12　冷却风扇系统照片

15.4.1　方案简介

车用冷却风扇主要用于发动机系统的散热，因而对于整个汽车系统来讲，冷却风扇是个很关键的部件。传统的冷却风扇都用的是直流有刷电机，但由于其电刷需要定期更换和维护等缺点，现在的冷却风扇慢慢地都在往 BLDC 上转。另外，由于冷却风扇的安装位置离发动机较近，因此对于器件的温度等级要求比较高，KEA 系列的温度范围为–40℃～125℃，可以满足该应用的需求。

该方案使用反电动势过零检测技术、带速度闭环和动态限流控制，同时具备过压、欠压和过流保护功能，能很好地满足车用冷却风扇的要求。

15.4.2　硬件结构

该车用冷却风扇系统采用飞思卡尔的 KEAZ128 作为其处理器，结合 MC33937A 三相桥驱动和 MC33903D 系统基础芯片实现 6 步无传感器的方波控制，控制原理图如图 15-13 所示，整个系统的框图如图 15-14 所示。三相无刷直流电机的相线通过分压网络连接到 KEAZ128 芯片的 ADC 模块的三个通道，用于三相反电动势的采集；母线电流则通过 MC33937A 内部的一个运放放大后再输入到 ADC 模块的一个通道，同时该电流也输入到 KEAZ128 的模拟比较器模块 ACMP0 用于过流的故障检测。母线电压通过分压网络输入到 ADC 模块的一个通道，同时也输入到模拟比较器 ACMP1 用于欠压故障的检测。也就是说，一旦发生欠压或过流事件，FTM2 的 6 路 PWM 模块就会被硬件异步关闭，从而保护系统。

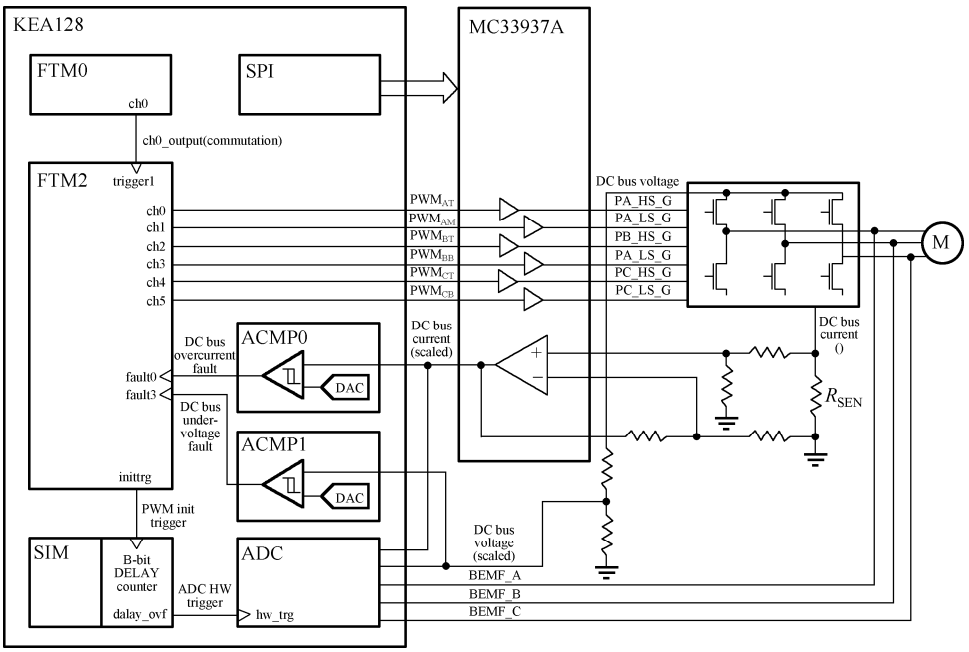


图 15-13 控制原理图

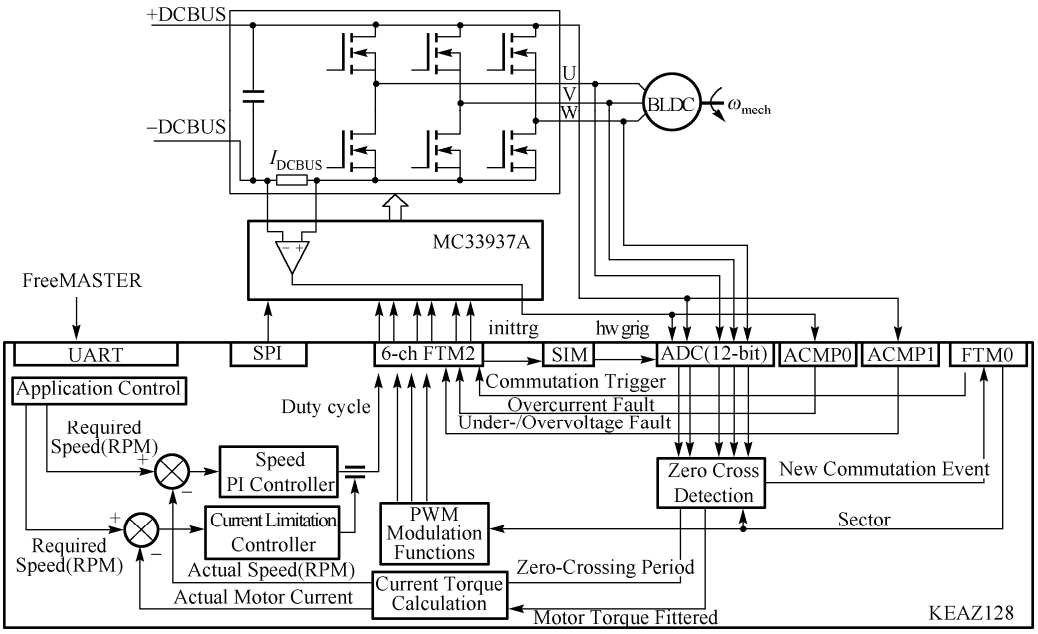


图 15-14 无传感器 BLDC 的系统框图



图 15-15 车用冷却风扇演示板实体图

KEAZ128 通过 SPI 来配置 MC33937A 这颗三相桥驱动芯片，其 FTM2 模块输出 6 路 PWM 到 MC33937A，用来驱动外部的 6 个 MOSFET。驱动板实体如图 15-15 所示。

MC33903D 系统基础芯片内有一个 LDO，其输出

5 V 给 KEA128 和板上的数字及模拟电路供电。同时 MC33903D 内有 LIN 和 CAN 的物理层接口，很容易实现 LIN 和 CAN 的通信。

最后需要提到一点，该演示板上还有一个刹车电路，可以用来泄放电机再生制动的能量，该电路由一个 MOSFET、一个续流二极管和一个外置的刹车电阻组成。

15.4.3　电气指标

车用冷却风扇演示板的电气指标如下。

（1）输入电压范围为 DC 8～18 V，典型值为 12 V。

（2）供电电流：V_{bat}=12 V 时电流的典型值为 56 mA。

（3）直流母线电压的分压系数为 1/5。

（4）直流母线采样电阻为 10 MΩ。

（5）直流母线电流采样放大倍数为 12。

（6）直流母线电流的测量范围为–20.833～20.833 A。

（7）直流母线过流保护点：20.83 A。

（8）刹车电阻阻值为 15 Ω。

（9）刹车电阻功率大小：最小为 3 W。

15.4.4　软件实现

软件的实现是按照前面所讲述的 BLDC 无传感器控制的方法来一步一步做的，为了更清晰地描述软件的实现，先给出该系统的应用流程、状态机的切换过程，以及应用时序和中断图，最后给出 BLDC 无传感器控制的核心过程。

1．系统应用流程

图 15-16 包含了两部分内容，一个是速度对时间的特性；另一个是应用的状态机和各种的过渡状态。软件中有三个主要的状态机：对齐、开环加速和闭环运行。在初始化外设完成后，软件就进入了对齐阶段，也就是将转子对齐到一个确定的位置。当对齐时间到了以后，软件就进入到开环加速阶段。当转速非常低的时候，反电动势电压非常小以至于不能检测到过零事件，所以电机需要在开环模式下加速一段时间。定子绕组产生的第一个矢量需要设定为相应的转子磁通矢量成 90° 的位置。开环加速的时间由开环换向的次数决定，开环换向的数量又取决于电机的机电时间常数、负载和所施加的电压。通常当开环加速到 5%的额定转速的时候，反电动势就已经足够大，从而可以很容易检测到其过零点。

2．状态机

软件中应用状态机是用一个指向状态函数的一维数组指针来实现的，这个数组的索引就指向了相应的应用状态函数。应用状态机由以下 7 个状态组成，其具体含义如下。

（1）INIT：配置初始化 PWM 的占空比、ADC 的外部触发和直流母线偏置校正，其指向 STOP 状态机。

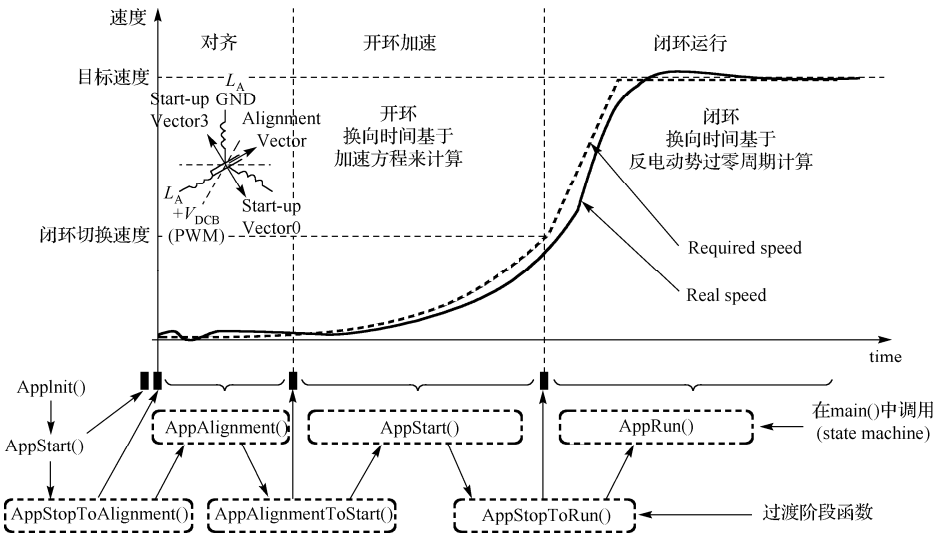


图 15-16　应用状态流程图

（2）CALIB：实现直流母线电压的校正，其指向 ALIGNMENT 状态机。

（3）ALIGNMENT：实现转子定位，将转子定位到一个固定位置，其指向 START 状态机。

（4）START：BLDC 换向是由开环控制，没有任何转子位置反馈。实际上就是开环加速阶段，将转子速度拉升到一个设定值，其指向 RUN 状态机。

（5）RUN：BLDC 是由无传感器算法进行速度闭环控制。当设置 appSwitchState 为 0 时，其指向 INIT。

（6）STOP：BLDC 停住，然后做好运行的准备。当设置 appSwitchState 为 1 时指向 CALIB 状态机。

（7）FAULT：故障检测是放到主循环中一直运行的，检测直流母线电压欠压、过压、直流母线过流和 MC33937A 故障。一旦任何故障被检测到，状态机会自动进入 FAULT 状态。PWM 输出被置为安全状态。要退出 FAULT 状态机，所有的故障源必须被清掉同时 faultSwitchClear 变量置 1 用来清除锁存的故障。然后状态机会自动进入 INIT 状态机。

各状态机可能的切换过程如图 15-17 所示。

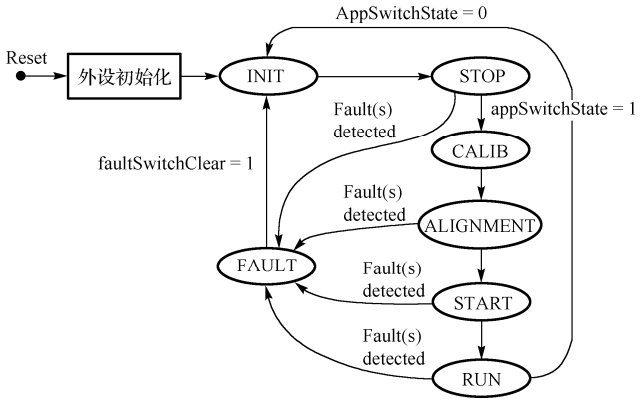


图 15-17　系统可能的状态切换图

3．应用时序和中断

如图 15-18 所示，灰色条纹代表的所执行的中断服务程序，下面的部分是对应的相电压波形。顶上第一行是表示当反电动势电压、母线电流和母线电压的 ADC 转换序列已经转换完成时所产生的 ADC 中断。在该中断里，FTM0 的计数值会被保存下来作为反电动势采集的参考点。过零检测算法是在发生换向事件后的每一个 ADC 转换完成中断中执行的，一旦过零被找到，过零检测算法就会停止，直到下一个新的换向事件到来。

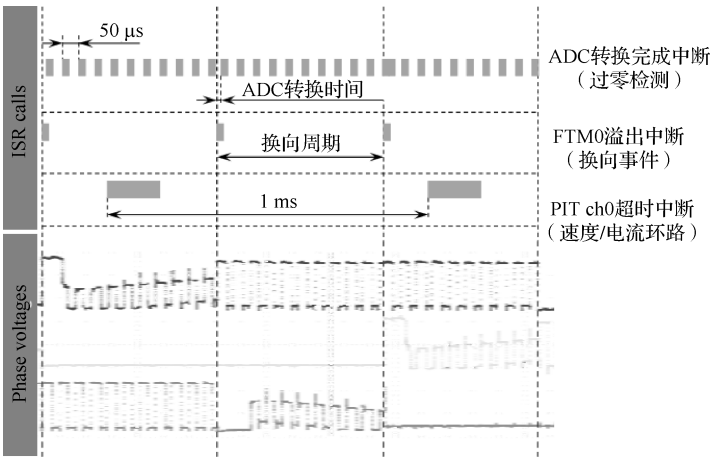


图 15-18　应用的时序和中断图

第二行是 FTM0 定时器溢出中断，用来产生换向事件。每次 FTM0 定时器溢出中断的间隔是由 BLDC 当前的速度决定的。这这个中断里，ADC 转换序列需要重新配置，因为每次换向事件后，反电动势检测哪一相是需要作相应调整的。

最后一行是每 1 ms 所产生的 PIT ch0 的超时中断。这个中断用于速度环路和电机的电流限制功能，实际上就是执行 PI 调节器的功能。

4．无传感器 BLDC 控制核心过程

无传感器 BLDC 控制的核心过程如下：过零检测模块比较反电动势和母线电压的一半以找到过零点。过零点发生后的 30° 电角度就会产生一个换向事件；该换向事件会去更改 PWM 的开关逻辑和更改过零检测模块的三相反电动势的选择逻辑，从而为下一次的换向事件做准备。过零检测模块会输出过零事件的周期和电机滤波后的电流，然后这两个值被送入到速度及电流计算模块。该模块会输出电机的实际速度及实际电流限制值，实际速度会和目标速度一起输入到速度 PI 调节器，调节器会输出一个电压，该电压值经过电流输出模块的限制后，会经过逆变器加载到 BLDC 上，从而实现可靠的无传感器的速度闭环控制。

15.4.5　开发中的相关工具

1．汽车数学和电机控制库 AUTOMATH_MCL

AUTOMATH_MCL（Automotive Math and Motor Control Library）是汽车数学和电机控制库的缩写，冷却风扇的应用实例就是使用该数学和电机控制库来实现的。AUTOMATH_MCL

可以简化开发难度和缩短开发周期，是电机控制开发工程师的强大武器，其结构层次如图 15-19 所示。

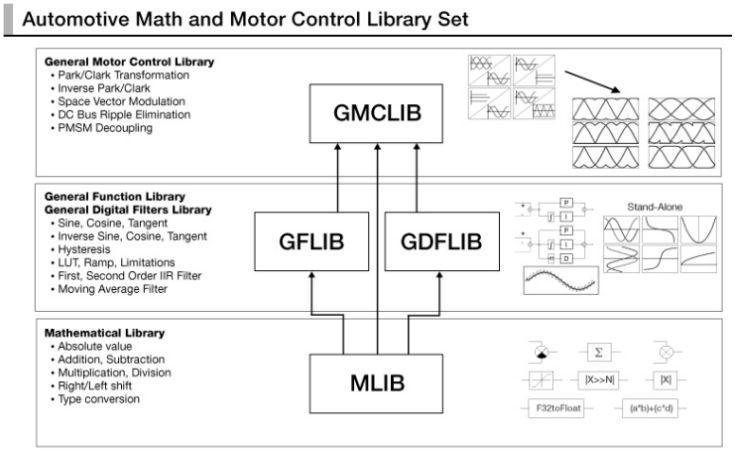


图 15-19　AMMCLIB 结构层次图

AUTOMATH_MCL 主要包含了以下几个层次。

（1）数学库 MLIB，里面包含了最基本的数学函数，如乘、加等运算。

（2）一般函数库 GFLIB，包含基本的三角和一般数学函数，像正弦运算 sine、余弦预算 cosine、斜坡函数 ramp 和 PI 调节器等。

（3）一般数字滤波函数 GDFLIB，包含了数字的 FIR 和 IIR 滤波。

（4）一般电机控制库 GMCLIB，包含了用于电机控制的标准算法，如 Clark/Park 变换、空间矢量调制模块 SVPWM 等。

读者可以根据自己的需要进行调用，非常方便实用。AUTOMATH_MCL 用于 KEA 的最新版本是 1.1.0，读者可以自行到 www.freescale.com/AutoMCLib 下载。

2．FreeMASTER

FreeMASTER 是飞思卡尔公司的一款实时调试工具，可用于实时监控应用变量。有了 FreeMASTER 这款强大的工具，调试就变得非常人性化了。客户可以通过设定直接 RS232 或者仿真工具来建立 FreeMASTER 和目标芯片的连接。连接成功后，用户可以跟踪变量，在线修改变量，使用图形界面显示变量变化波形等功能。这些功能对于电机控制的调试来讲，非常实用。例如，按一个按钮或者改变状态机为 Start，电机就可以跑起来了，同时还可以监控电机的状态机、转速、电压电流大小、板子温度等重要信息；实时波形工具配置好后还可以看到电机的三相相电流波形，电机的反电动势波形等。用户还可以定制自己的用户界面，非常的灵活。图 15-20 就显示了 FreeMASTER 的打开一个工程后的界面，关于更多应用上的信息可以登录 http://www.freescale.com/freemaster 进行查看。

3．电机控制应用调试工具 MCAT

电机控制应用调试工具 MCAT（Motor Control Application Tuning Tool）是一款非常直观

且界面友好的调试工具，如图 15-21 所示，该工具由电机选择工具条、选项卡菜单和工作区组成。

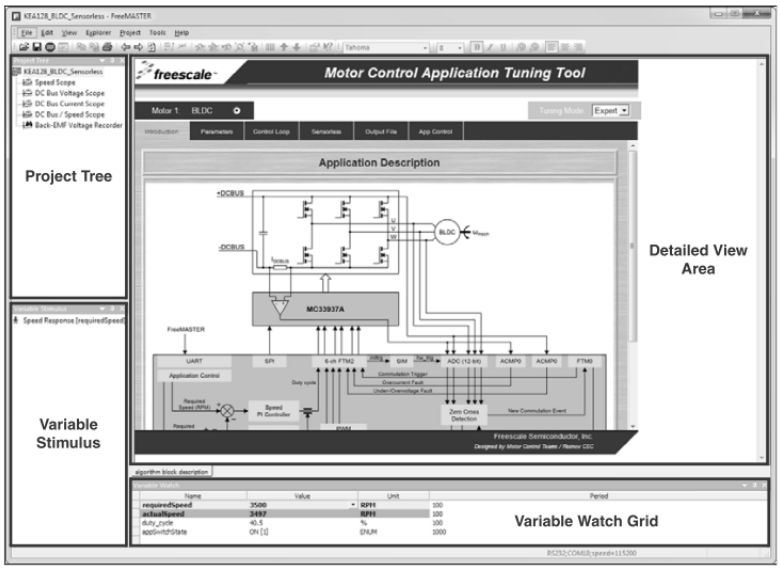


图 15-20　打开一个工程后的 FreeMASTER 的界面

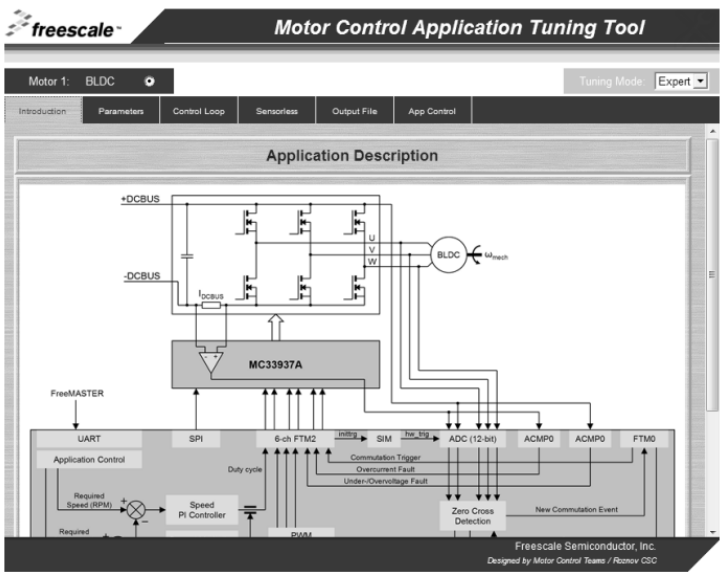


图 15-21　MCAT 界面

MCAT 工具支持输出控制算法所需的常量头文件，同时也可以在线更新应用控制的参数，如控制环路、速度斜坡等。只需要单击每个控制选项卡上的“Update Target”按钮就可以更新这些变量，另外这些设置参数可以保存成 MCAT 文件，以便于下次轻松装载。

MCAT 的目标是引导用户通过一步一步地调节参数，用自己的板子来运转自己的电机，达到自己想要的性能。目前 MCAT 已经可以很好地支持 KEA 系列产品，关于 MCAT 更多的信息，请登录 http://www.freescale.com/mcat。

15.4.6 方案总结

车用冷却风扇的实例讲述了 BLDC 无传感器控制的一些要点，从硬件架构的考虑到电气指标，从应用流程图、状态机再到无传感器控制的核心思想，最后还谈到了飞思卡尔关于电机控制非常有用的一些工具等。由于篇幅的限制，很多细节未能一一罗列，更多的信息读者可以参考飞思卡尔的 DRM151 及其相应的参考软件包。

15.5 关于车用 BLDC 的展望

可以看到的趋势是，BLDC 在汽车的应用会越来越多。风机、泵和压缩机会先行一步，首先步入到 BLDC 的队伍中来。

BLDC 反电动势是梯形波的，可以采用方波控制策略，而对于反电动势是正弦波的，也叫作 PMSM，就需要更高级的控制策略，如 FOC 控制等。目前对于汽车里的风机、泵和压缩机控制，绝大部分情况都是无传感器控制，如果采样方波控制，KEA 产品会是一个很好的选择，而对于 FOC 控制而言，另一个系列 S12ZVM 则非常适合，用户可以根据自己的需要进行选择。

第 16 章

有关问题的进一步讨论

本章导读：本章阐述 KEA128 的位带技术及位操作引擎技术。主要内容有：16.1 节讲解位带技术及应用方法，16.2 节介绍位操作引擎技术及应用方法。每节中，先简单介绍相关技术，然后给出相应技术的应用机制解析，最后还给出使用注意事项。

本章阅读资料：本章主要参考的是 KEA128 参考手册中的第 4 章（Memory Map）、第 17 章（BME）相关的章节。

16.1 位带技术及应用方法

在 MCU 编程中，通常情况下，对 RAM 及外设寄存器的操作，只能进行整字节读取/写入（Load/Store），对一位的操作需特殊处理。以 Cortex-M0+为内核的 KEA128 微控制器内部具有“位带”机制，可以支持利用“位带别名区”的 98304 字地址编号实现对 12 KB 的 SRAM_U 的 98304 个位[①]的操作。

16.1.1 位带别名区概述

KEA128 微控制器的 SRAM（16 KB）分为两部分：SRAM_L（4 KB）和 SRAM_U（12 KB）。SRAM_U 有两种途径支持位操作，分别是位带别名区与位操作引擎。本节阐述位带别名区，16.2 节阐述位操作引擎。针对位带别名区的 32 位字地址的写操作与 SRAM_U 上的目标位的“读-改-写”操作作用相同，但是只有一个周期。根据 KEA128 参考手册得知，12 KB 的 SRAM_U 位带区（Bit-Band region）地址为 0x2000_0000-0x2000_2FFFF，其所对应的 384 KB 的位带别名区（Alias bit-band region）地址为 0x2200_0000-0x2205_FFFF。位带别名区只支持简单的置 1 和清 0 操作，目的是方便对位的操作。

16.1.2 位带别名区的应用机制解析

我们来看看，一般情况下，如何修改内存中的某一位，基本要求是改动一位，不能影响其他位。以 32 位字长为例进行解析，设目标地址为 0x20002FF0，改动其第 16 位为“0”的方法如下。

（1）读一个字：读出 0x20002FF0～0x20002FF3 中内容到变量 temp 中。

① 位带别名区地址空间（0x2200_0000～0x2205_FFFF，384 KB，98304 字）对应 SRAM_U 位带区（0x2000_0000～0x2000_2FFFF，12 KB）的 12×1024×8=98304。

```
temp=(*( volatile unsigned long int *)(unsigned long int)0x20002FF0);
```

（2）改一个位：将 temp 中的第 16 位清 0。

```
temp=temp&(0xFFFEFFFF);
```

（3）写一个字：将 temp 写回目标地址。

```
(*(volatile unsigned long int *)(unsigned long int)0x20002FF0)=temp;
```

这就是通常所说的“**读-改-写**”操作，即读内存赋给临时变量，然后对临时变量进修改，最后将临时变量结果写回内存。

如果我们使用 KEA128 微控制器硬件机制所提供的位带别名区将 SRAM_U 位带区地址为 0x20002FF0 内存单元第 16 位变成“0”，仅需一步写操作就可以实现。

```
(*(volatile unsigned long int *)(unsigned long int)0x2205FE40)=0;
```

其中位带别名区的地址 0x2205FE40 可以通过公式 0x22000000+(0x20002FF0–0x20000000)×32+16×4 计算得到。

为了充分说明位带技术的优势，我们在 KDS3.0.0 编译环境中对上述代码反汇编进行对比，可以发现使用位带别名区技术比原有“读-改-写”方法的代码空间要小，执行效率更高。通过“位带别名区”的写操作就可以实现对 SRAM_U“位带区”中位的操作，将位操作的“读-改-写”过程就变为只有“写”的操作，提高了程序的运行效率。

一般情况下的机器码如下。

```
//SARM_U 位带区读一个字
temp=(*( volatile unsigned long int *)(unsigned long int)0x20002FF0);
 80e:   4b36        ldr r3, [pc, #216]  ; (8e8 <main+0xe8>)
 810:   681b        ldr r3, [r3, #0]
 812:   60fb        str r3, [r7, #12]
//改一个位
temp=temp&(0xFFFEFFFF);
 814:   68fa        ldr r2, [r7, #12]
 816:   4b35        ldr r3, [pc, #212]  ; (8ec <main+0xec>)
 818:   4013        ands    r3, r2
 81a:   60fb        str r3, [r7, #12]
//SARM_U 位带区写一个字
(*( volatile unsigned long int *)(unsigned long int)0x20002FF0)=temp;
 81c:   4b32        ldr r3, [pc, #200]  ; (8e8 <main+0xe8>)
 81e:   68fa        ldr r2, [r7, #12]
 820:   601a        str r2, [r3, #0]
```

使用位带别名方法的机器码如下。

```
//位带别名区写一个字
(*( volatile unsigned long int *)(unsigned long int)0x2205FE40)=0x00000000;
 822:   4b33        ldr r3, [pc, #204]  ; (8f0 <main+0xf0>)
 824:   2200        movs    r2, #0
 826:   601a        str r2, [r3, #0]
```

16.1.3　位带别名区使用注意点

1．使用的 RAM 区域

SRAM_U 作为普通内存区时，一般被用来存储全局变量、临时变量（堆栈空间）等。当 SRAM_U 作为位带区时，我们仅需使用其中很小一部分区域作为位带区，其余大部分仍然用于普通内存区。如果地址随意使用，位带区和普通内存区会出现重叠，从而出现错误。为此，我们一般在链接文件中修改堆栈大小和堆栈 SP 指针，将堆栈的最高地址向小端移动，留出部分地址空间供 SRAM_U 位带区使用。

2．计算位地址的方法

KEA128 微控制器硬件提供位带别名区对 SRAM_U 位带区对应位的置 1 或清 0 操作。位带别名区操作实质是一种内存映射关系，系统将 SRAM_U 位带区的存储单元按位映射到对应位带别名区的 32 位字上，位带别名区中的一个 32 位地址，对应位带区一个地址中的一个位。按字访问位带别名区的存储单元时，就相当于访问位带区对应位。通过对位带别名区地址的访问等同于对真实地址的某个位的访问。位带区与位带别名区的映射关系如图 16-1 所示。

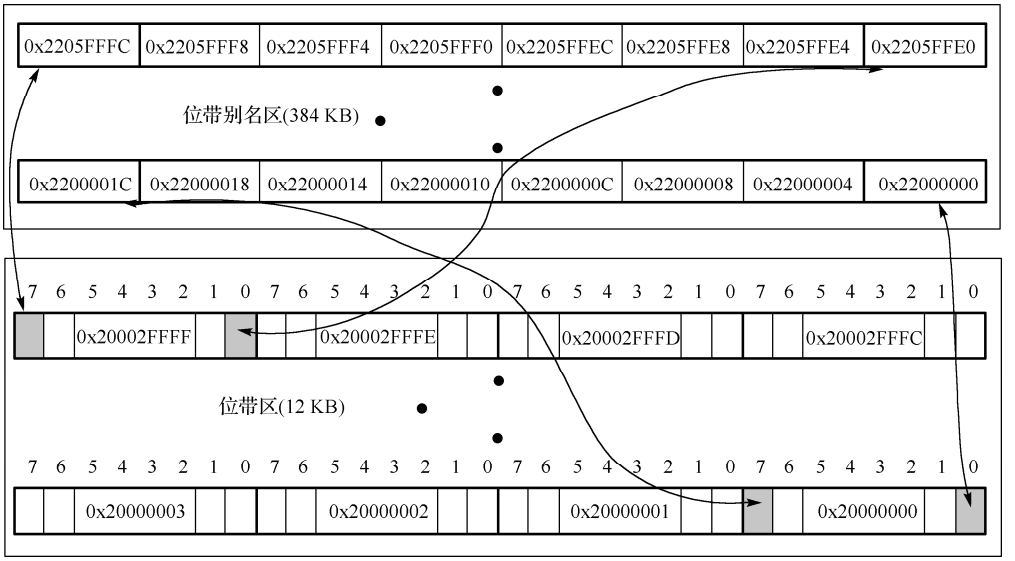


图 16-1　位带区和位带别名区的映射关系

通过位带技术映射存储空间之后，编写程序时可以方便的实现位操作。位带别名区的第 0 位的值决定写入目标位的值，为 1 代表向目标位写 1；为 0 代表目标位清 0。可以通过向位带别名区写 0x00000000，表明目标位清 0；写 0x00000001 表明目标位置 1。根据 KEA128 微控制器位带区“位”与位带别名区“字”的对应关系，位带别名区地址的计算方法为

位带别名区地址 = 位带别名区基地址 + 位带区字节偏移量×32 + 位偏移量×4

设 SRAM_U“位带区”的地址为 X，需要置 1 或清 0 的位为 m（$0 \leqslant m \leqslant 31$），可以通过公式计算出位带别名区的地址为

$$0x22000000+(X-0x20000000)\times 32+m\times 4$$

3．注意使用 volatile 关键字

此外，在 C 语言编程设计中使用位带技术时，所访问的存储器单元变量必须使用关键字 volatile 来加以定义，用于规定 C 编译器不允许对其限定的变量进行优化处理。编译后的程序每次需要存储或读取这个变量的时候，都会直接从变量地址中读取数据。如果没有 volatile 关键字，则编译器可能优化读取和存储，可能暂时使用寄存器中的值，如果这个变量由别的程序更新了的话，将出现不一致的现象。

16.2 位操作引擎技术及应用方法

16.2.1 位操作引擎概述

位操作引擎（Bit Manipulation Engine，BME）对基于 Cotex-M0+微控制器的 SRAM_U 和外设地址空间内存的原子“读-改-写”操作提供了硬件支持，为 SRAM_U 和外设地址数据的读写操作提供了一种封装地址方案，BME 的封装参数可以从系统总线事务中上获得，由处理器内核产生。

BME 提供的封装式存储包含有 3 个逻辑操作（AND、OR、XOR）和一个数据位插入操作。对于这些操作来说，BME 是把一个单独的封装式 AHB 存储事务处理过程转换成一个包含 2 个周期的原子级“读-改-写”操作序列。在第一个 AHB 数据序列中包含了读-改操作，在第二个数据序列中包含了写操作。

BME 提供的封装式载入有如下功能：两个单独的置位和清除操作、无符号数据位提取操作。对于置位和清除操作来说，BME 是把一个单独的封装式 AHB 存储事务处理过程转换成一个包含 2 个周期的原子级“读-改-写”操作序列，在第一个 AHB 数据序列中包含了读-改操作，在第二个数据序列中包含了写操作，然后读出的原始数据返回到处理器内核中。对无符号位提取操作来说，在数据被提取时，封装式载入事务处理操作是停滞的。在第二个 AHB 数据序列中，返回处理器。该操作只是一个简单的数据读取操作，不是读-改-写操作。

BME 提供的封装式存储和载入不同功能所对应的外设地址的各个位含义如表 16-1 所示，其中 AND、OR、XOR、BFI 四种操作为封装地址写操作，LAC1、LAS1、UBFX 为封装地址读操作。

表 16-1 封装式存储和载入功能外设地址各位含义表

<table>
<tr><th>功 能</th><th>位 31</th><th>位 30～29</th><th>位 28</th><th>位 27～26</th><th>位 25～21</th><th>位 20</th><th>位 19～0</th></tr>
<tr><td>逻辑与 AND</td><td>0</td><td rowspan="5">10：外设
01：SRAM_U</td><td>0</td><td>01</td><td rowspan="3">未使用</td><td rowspan="5">未使用</td><td rowspan="5">SRAM_U 或外设偏移地址</td></tr>
<tr><td>逻辑或 OR</td><td>0</td><td>0</td><td>10</td></tr>
<tr><td>逻辑异或 XOR</td><td>0</td><td>0</td><td>11</td></tr>
<tr><td>清除 1 位 LAC1</td><td>0</td><td>0</td><td>10</td><td rowspan="2">b：位标识符</td></tr>
<tr><td>置位 LAS1</td><td>0</td><td>0</td><td>11</td></tr>
<tr><th>功 能</th><th>位 31</th><th>位 30～29</th><th>位 28</th><th colspan="2">位 27～23</th><th>位 22～19</th><th>位 18～0</th></tr>
<tr><td>数据位插入 BFI</td><td>0</td><td rowspan="2">10：外设
01：SRAM_U</td><td>1</td><td colspan="2" rowspan="2">b：位标识符</td><td rowspan="2">w：位宽度</td><td rowspan="2">SRAM_U 或外设偏移地址</td></tr>
<tr><td>无符号位提取 UBFX</td><td>0</td><td>1</td></tr>
</table>

根据 KEA128 参考手册得知，外设地址空间占了 516 KB：基址是 0x4000_0000 的 512 KB 准外设地址空间和基址是 0x400F_F000 的 4KBGPIO 访问空间。封装地址空间是一个 448 MB 的区域，范围为 0x4400_0000～0x5FFF_FFFF。与 SRAM_U 关联的封装地址空间也是一个 448 MB 的区域，范围为 0x2400_0000～0x3FFF-FFFF。

16.2.2　位操作引擎的应用机制解析

下面，我们来介绍利用位操作引擎 BME 技术对 SRAM_U 和外设地址的操作。

1．位操作引擎操作 SRAM_U

同样，以 32 位字长为例进行解析，设 SRAM_U 目标地址为 0x20002FF0，改动其第 16 位，变为“0”。可以利用清除 1 位操作和 AND 操作方法来实现。

（1）利用 BME 的清除 1 位操作方法。根据下表可以编写对应外设封装地址。

位 31	位 30～29	位 28	位 27～26	位 25～21	位 20	位 19～0
0	01	0	10	10000	0	0000 0010 1111 1111 0000

```
addr[31:0]=0010 1010 0000 0000 0010 1111 1111 0000=0x2A002FF0
U32temp=(*( volatile unsigned long int *)(unsigned long int) 0x2A002FF0);
```

读取该外设封装地址，即完成了清除 1 位操作。

（2）利用 BME 的 AND 操作方法。根据下表可以编写对应外设封装地址。

位 31	位 30～29	位 28	位 27～26	位 25～21	位 20	位 19～0
0	01	0	01	00000	0	0000 0010 1111 1111 0000

```
addr[31:0]=0010 0100 0000 0000 0010 1111 1111 0000=0x24002FF0
(*( volatile unsigned long int *)(unsigned long int) 0x24002FF0)=0xFFFEFFFF;
```

写 0xFFFFEFFF 给该外设封装地址，即完成与运算操作，实现第 16 位清 0。

2．位操作引擎操作外设地址空间

对于外设寄存器的操作，和 SRAM_U 的方法类似。根据手册得知，KEA128 微控制器 PORTC0 是 GPIOA 寄存器的第 16 位，其地址为“*(volatile unsigned long int *)(unsigned long int) 0x400FF000”。我们以 PORTC0 为例，将输出寄存器的第 16 位清 0，使 PORTC0 引脚输出低电平。可以利用清除 1 位操作方法和 AND 操作方法。

利用通常所说的“读-改-写”操作，对外设地址空间的访问操作和 16.1 节中阐述 SRAM_U 内存访问操作一样，此处不再赘述。

（1）利用 BME 的清除 1 位操作方法。根据下表可以编写对应外设封装地址。

位 31	位 30～29	位 28	位 27～26	位 25～21	位 20	位 19～0
0	10	0	10	10000	0	1111 1111 0000 0000 0000

```
addr[31:0]=0100 1010 0000 1111 1111 0000 0000 0000=0x4A0FF000
U32temp=(*( volatile unsigned long int *)(unsigned long int) 0x4A0FF000);
```

读取该外设封装地址，即完成了清除 1 位操作。

（2）利用 BME 的 AND 操作操作方法。根据下表可以编写对应外设封装地址。

位 31	位 30～29	位 28	位 27～26	位 25～21	位 20	位 19～0
0	01	0	01	00000	0	1111 1111 0000 0000 0000

```
addr[31:0]=0100 0100 0000 0000 0010 1111 1111 0000=0x440FF000
(*( volatile unsigned long int *)(unsigned long int) 0x440FF000)=0xFFFEFFFF;
```

写 0xFFFEFFFF 给该外设封装地址，即完成与运算操作，实现第 16 位清 0。

我们在 KDS3.0.0 编译环境中对上述代码反汇编进行对比，可以发现使用 BME 技术比原有“读-改-写”方法的代码空间要小，执行效率更高。

一般情况下的机器码如下。

```
//SARM_U 位带区读一个字
temp=(*( volatile unsigned long int *)(unsigned long int)0x20002FF0);
 80e:   4b36        ldr r3, [pc, #216]  ; (8e8 <main+0xe8>)
 810:   681b        ldr r3, [r3, #0]
 812:   60fb        str r3, [r7, #12]
//改一个位
temp=temp&(0xFFFEFFFF);
 814:   68fa        ldr r2, [r7, #12]
 816:   4b35        ldr r3, [pc, #212]  ; (8ec <main+0xec>)
 818:   4013        ands    r3, r2
 81a:   60fb        str r3, [r7, #12]
//SARM_U 位带区写一个字
(*( volatile unsigned long int *)(unsigned long int)0x20002FF0)=temp;
 81c:   4b32        ldr r3, [pc, #200]  ; (8e8 <main+0xe8>)
 81e:   68fa        ldr r2, [r7, #12]
 820:   601a        str r2, [r3, #0]
//*************************************************************
//外设地址读一个字
temp=(*( volatile unsigned long int *)(unsigned long int)0x400FF000);
 850:   4b2a        ldr r3, [pc, #168]  ; (8fc <main+0xfc>)
 852:   681b        ldr r3, [r3, #0]
 854:   60fb        str r3, [r7, #12]
//改一个位
temp=temp&(0xFFFEFFFF);
 856:   68fa        ldr r2, [r7, #12]
 858:   4b24        ldr r3, [pc, #144]  ; (8ec <main+0xec>)
 85a:   4013        ands    r3, r2
 85c:   60fb        str r3, [r7, #12]
//外设地址写一个字
(*( volatile unsigned long int *)(unsigned long int)0x400FF000)=temp;
 85e:   4b27        ldr r3, [pc, #156]  ; (8fc <main+0xfc>)
 860:   68fa        ldr r2, [r7, #12]
 862:   601a        str r2, [r3, #0]
```

使用 BME 方法的机器码如下。

```
//BME 操作 SAM_U
```

```
//BME 清除一位
U32temp=(*( volatile unsigned long int *)(unsigned long int) 0x2A002FF0);
 844:   4b2b        ldr r3, [pc, #172]  ; (8f4 <main+0xf4>)
 846:   681b        ldr r3, [r3, #0]
 848:   60bb        str r3, [r7, #8]
//BME 的 AND 操作
(*( volatile unsigned long int *)(unsigned long int) 0x24002FF0)=0xFFFEFFFF;
 84a:   4b2b        ldr r3, [pc, #172]  ; (8f8 <main+0xf8>)
 84c:   4a27        ldr r2, [pc, #156]  ; (8ec <main+0xec>)
 84e:   601a        str r2, [r3, #0]
//BME 操作外设寄存器
//BME 清除一位
U32temp=(*( volatile unsigned long int *)(unsigned long int) 0x4A0FF000);
 864:   4b26        ldr r3, [pc, #152]  ; (900 <main+0x100>)
 866:   681b        ldr r3, [r3, #0]
 868:   60bb        str r3, [r7, #8]
//BME 的 AND 操作
(*( volatile unsigned long int *)(unsigned long int) 0x440FF000)=0xFFFEFFFF;
 86a:   4b26        ldr r3, [pc, #152]  ; (904 <main+0x104>)
 86c:   4a1f        ldr r2, [pc, #124]  ; (8ec <main+0xec>)
 86e:   601a        str r2, [r3, #0]
```

16.2.3　位操作引擎对 GPIO 部分的使用说明

外设地址空间占用 516 KB，即 512 KB 基于 0x4000_0000 加上 4 KB 基于 0x400F_F000 空间的 GPIO 访问，这种内存布局主要是为了兼容 Kineti K 系列 MCU。

GPIO 地址空间被硬件多重映射，它出现在“标准”系统地址 0x400F_F000 和在物理上位于与槽对应的地址为 0x4000_F000，使用 addr[19]进行封装操作。对于 AND、OR、XOR、LAC1 和 LAS1，这个位作为真实的地址位；对于 BFI 和 UBFX，这个位定义了 w 字段说明符的最低有效位。

因此，直接对 GPIO 的访问操作和封装的 AND、OR、XOR、LAC1 和 LAS1 操作可以使用标准 0x400F_F000 基地址，然而 BFI 和 UBFX 封装操作必须轮流使用 0x400F_F000 基地址。当然可以简单地使用 0x400F_F000 作为所有未封装的 GPIO 访问和把 0x4000_F000 作为所有封装访问的基地址。UBFX 和 BFI 一样操作的都是映射内存空间，用来操作 GPIO 时要以 F000 为起始地址。具体访问说明见表 16-2。

表 16-2　外设封装和 GPIO 地址细节

外设地址空间	说　明
0x4000_0000～0x4007_FFFF	未封装（正常）外设访问
0x4008_0000～0x400F_EFFF	非法地址；试图引用会失败并错误终止
0x400F_0000～0x400F_FFFF	使用标准地址进行未封装（正常）GPIO 访问
0x4010_0000～0x43FF_FFFF	非法地址；试图引用会失败并错误终止
0x4400_0000～0x4FFF_FFFF	封装 AND、OR、XOR、LAC1、LAS1 引用外设，GPIO 基于 0x4000_F000 或 0x400F_F000
0x5000_0000～0x5FFF_FFFF	封装 BFI、UBFX 引用外设，GPIO 只基于 0x4000_F000

16.2.4 位操作引擎使用注意点

1．数据位宽度

上述范例中仅以 32 位字的操作为例进行了说明，BME 对于各种操作均支持 8、16、32 位宽度的数据。对于 32 位宽度模式中，表 1 中的 SRAM_U 或外设地址 mem_addr 的位[1:0]应为 00；对于 16 位宽度模式中，SRAM_U 或外设地址 mem_addr 的位[0]应为 0；8 位宽度模式中，地址没有类似限制。

2．w1c 位的操作

BME 执行的是读-修改-写操作，而很多寄存器有些位是 w1c，也就是所谓的 write-1-clear，写 1 清 0 的工作方式。使用 BME 时就需要特别注意和小心了，否则会出现很多不可预料的后果。 如果一个寄存器中有多个连续的 w1c 位，我们就不要使用 LAS1 来对寄存器写 1 清 0 了，因为在 LAS1 这个操作中，其中有一步操作是将数据读回（在 KEA128 参考手册中提到“read data return to core”）。这一步会将原本不需要清 0 的位给清了，所以 BME 用于处理 w1c 位时要特别小心。

3．注意使用 volatile 关键字

同样在使用 BME 技术时，所访问的存储器单元变量也必须使用关键字 volatile 来加以定义。

16.2.5 测试实例

测试工程位于网上光盘中的“..\ KEA128-Bit-Band&BME”文件夹，功能是利用多种方法对 KEA128 板上 Light_0 灯所对应的 GPIO 引脚控制。此外，可以通过 KDS3.0.0 在线调试方法，观测利用位带技术和 BME 技术对 SRAM_U 内存区的操作过程。

16.3 EMC 问题的探讨

16.3.1 简介

为了使汽车电子产品能够在苛刻的环境中毫无差错地运行，并且不造成干扰，EMC 的问题变得越来越关键，并成为产品可靠性的一个重要因素。本节针对读者在使用 KEA 系列产品过程中如何考虑 EMC 问题给出了一些提示，主要针对硬件设计、PCB 布局布线和软件设置做了探讨，以帮助读者在设计阶段就增强其产品的 EMC 性能。

16.3.2 硬件设计

硬件设计主要是考虑板级的设计，包括 PCB 的布局布线和各种不同类型 IO 口的处理。其首要原则是利用 EMC 方面的知识来阻止可能影响系统运行和稳定性的任何内部与外部噪声，最大程度地降低噪声源对 MCU 造成的耦合影响，减少干扰源的噪声量，以及提高系统的抗扰度。

PCB 布局是关乎从内部或外部噪声源是否会耦合噪声的关键因素，而布局中的布线则充当了耦合路径，布线的长度、宽度、形状和位置等几何因素会严重影响耦合的效果。合理的布局布线有助于将噪声源和系统分离，并提高系统的抗扰度。下面将分章节介绍可靠硬件设计的一些技巧。

1．器件布局

器件布局的一般准则如下。

（1）将螺孔和安装点的所有位置标记为禁用区。

（2）根据固定位置的要求来布局所有用户接口组件（如显示面板、按钮开关和连接器）。

（3）将高功率电路与低功率和噪声敏感电路分隔开来。

（4）将器件关联成组，并尽量将各组按照符合相关的信号流流向的逻辑顺序排列。

（5）标识需要布局在 MCU 附近的所有关键组件，以及从 MCU 输入端口连接到电源或接地的外部组件（如电源退耦电容及输入信号滤波组件）。

（6）最小化电源环路和接地环路面积。

（7）减少电源和 MCU 接地之间的共模阻抗。

2．电源和接地布线

电源和接地平面的 PCB 布局对于板级 EMC 性能及其重要，尤其是在使用 5 V 和 12 V 的多电源系统中。可以使用 PCB 布局技术将接地平面划分成两个部分，其中一个部分定义为 12 V 电路的返回路径，另一个部分定义为 MCU 和其他关键组件的 5 V 返回路径，这样，12 V 接地的噪声就不会通过地线和 5 V 接地噪声耦合了。

PCB 布局中 MCU 接地的连接方法是保障 EMC 性能的关键因素。建议在 MCU 下方填充一个接地平面，并将所有 VSS 引脚连接在一起，这样就确保所有 MCU 的 VSS 引脚保持在相同的点位，同时可以最小化从 MCU 到旁路电容的电流返回路径上的电感。对于 LQFP 封装，可以进一步将 MCU 接地平面扩展到封装拐角点，以实现较短的接地路径，并最大限度地减少围绕 MCU 的外设组件的环路面积。

3．退耦和旁路

退耦和旁路实际上是两个概念，但很容易混淆。

退耦用于隔离公用线上电路间的噪声，如电源线就是从稳压器到 MCU 的公用线之一。

旁路是通过一个旁路电容来分流一个阻抗路径，以减少在该路径中的高频电流波动。

为 MCU 添加退耦和旁路电容的效果很大程度上取决于连接位置和顺序，PCB 布局中 MCU 电源引脚（VDD 和 VSS）的准则如下。

（1）将电源引出的电源和接地线依次连接到退耦电容、旁路电容，以及 MCU 的 VDD 和 VSS 脚。

（2）为了使得电源的环路面积最小，并行排布电源和接地线。

（3）将旁路电容尽量靠近 MCU 每个电源对（VDD-VSS）位置排布。

4．晶体振荡器电路

连接到单片机的时钟引脚（EXTAL 和 XTAL）的晶体振荡器组件对外部噪声十分敏感。PCB 以保护环的形式排布接地线，这样可以最大程度减少接入晶振电路的噪声，一般准则如下。

（1）不要在晶振电路附近或横跨电路底侧排布任何信号路径（除地线外）。

（2）将晶振电路组件（晶振、反馈电阻和负载电容）尽量排布在靠近 EXTAL 和 XTAL 引脚的位置。

（3）如果对时钟精度要求不是特别高时，可以选择内部振荡器作为时钟源以提高 EMC 性能。

（4）如果使用双层或多层 PCB，请将负载电容的接地直接连到接地平面。

（5）选择满足系统要求的最小总线频率。

（6）在振荡器电路上采用最小走线长度。

（7）使用合理参数值的反馈电阻和负载电容。

5．间距和隔离

如果在同一块电路板上，高功率电路与低功率电路并列铺排，则隔离每个电路模块就很有必要。在某些情况下，如果电路板的尺寸有限，则可能需要添加物理槽以提供更好的隔离。同样，出于 ESD（静电释放）方面的考虑，需要在 PCB 走线与安装螺孔或者板边沿之间保留足够的间距。

6．输入输出端口

与输出功能相比，配置为输入功能的 MCU I/O 端口对噪声更为敏感。建议给每个输入功能引脚添加一个 RC 滤波器，用于衰减外部噪声源注入到引脚中的噪声，该滤波器的位置应靠近引脚，RC 滤波器的取值取决于输入信号及其特征（数字或模拟，以及变化率）。串联电阻的典型值在 100 Ω到 1 kΩ的范围内，而滤波电容的典型值在 1000 pF～0.1 μF。

RESET_b 和 NMI_b 是 KEA 中的特殊引脚，其可以当做电源引脚来做滤波处理，也就是说，需要最小化其电容的接地环路及上拉电阻的 VDD 环路。

不要将未使用的 I/O 引脚连接到任何元件，应该将它们悬空，并在软件中设置为低电平输出。定期刷新引脚状态，以避免噪声引起其状态变化。如果特定应用中不允许悬空引脚，那么就把该引脚连接一个 10 kΩ的下拉电阻。不要将任何没有用到的 I/O 引脚直连到电源或地。

16.3.3 软件设计

在充分考虑 EMC 要求的条件下进行合理的软件设计可以改善噪声环境中的系统整体性能和提高工作的稳定性。

一般而言，软件设计无法改变系统中耦合噪声的物理媒介，或者减少外部源产生的噪声绝对值。但是软件可以提供一些方法让用户在故障状态下选择纠正措施，以及实施预防性功能来实现系统保护。建议采用以下软件技巧来实现良好的防御性软件设计。

（1）启用看门狗功能以避免代码失控。

（2）定期刷新数据方向设置寄存器。

（3）填充未使用的内存以避免代码失控。

（4）定义所有中断向量，即使有些向量未使用。

（5）选择锁频环（FLL）啮合模式，可以抑制瞬态噪声干扰。

（6）多次读取每个边沿触发中断服务的输入数据，以确认输入事件是否有效。

（7）在输入端口上启用数字滤波器。

KEA 自身强大的芯片级别的 EMC 性能加上本节介绍的应用级的 EMC 技巧，会使得读者设计的产品在复杂的噪声环境里，稳定且可靠地工作。

16.4　基于 CAN 接口及 Bootloader 的程序更新方法

16.4.1　概述

在一些特定场合，很多应用或者产品都需要升级固件，以便修复某些发现的 Bug 或者提高性能。其中大多数应用或者产品都不使用专用的调试接口，而是使用 UART、IIC、CAN 等通信接口。在汽车应用中，CAN 通信最为广泛，因此本节讲的就是如何使用 CAN 通信接口来更新程序的方法。CAN Bootloader 是通过 CAN 通信协议将新的应用程序烧录到单片机 Flash 的过程。在网站上本章节部分可以找到实现 CAN Bootloader 的程序包（包含 CAN Bootloader 程序和三个应用程序及其 S19 文件）和上位机软件 DemoCAN，接下来请参考具体操作指南来实现这个功能。

16.4.2　操作指南

1．系统连接

图 16-2 为系统连接示意图，需要一个 USB 转 CAN 的工具，将 TRK-KEA128 板上的 CANH 和 CANL 连接到 USB 转 CAN 工具上；同时注意用户需要将 TRK-KEA128 板上 R66（未焊接）的两个空焊盘进行短接。

2．固件更新流程

（1）开发板烧写 Bootloader 程序：先用 CodeWarrior10.6 打开 CAN Bootloader 工程，编译好后写入开发板。

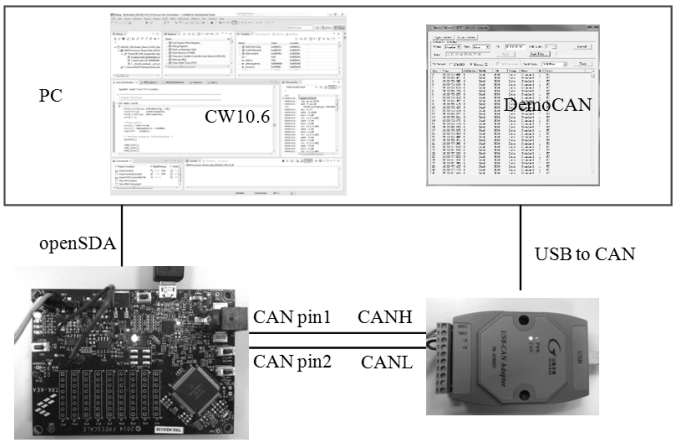


图 16-2 系统连接示意图

（2）连接 DemoCAN 上位机：在 PC 端打开 DemoCAN 上位机软件，选择波特率 1000 kbps，同时确保为标准帧，帧 ID 为“00 00 00 64”。然后单击“open device”启用设备，确保 USB-CAN 工具开始工作。接着单击“Send File”选择一个 S19 文件，软件包中有三个应用例程，可以随便选择一个；这时软件就开始通过 CAN 总线发送“FF”，直到接收到 KEA128 送过的反馈，如图 16-3 所示。

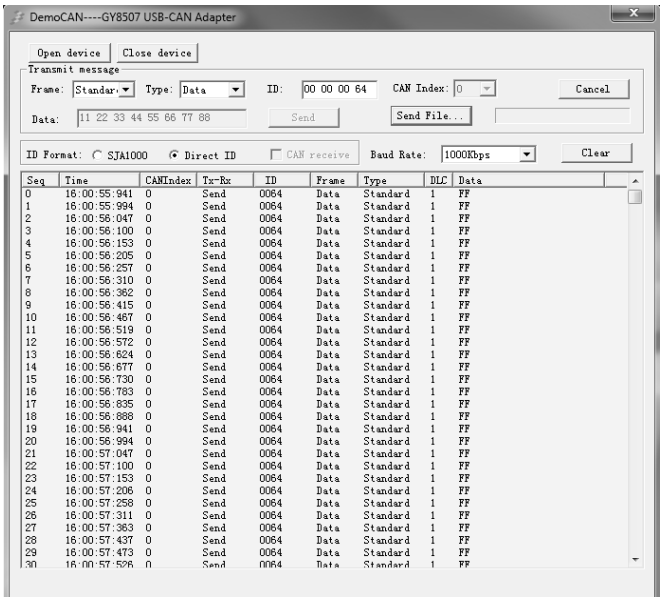


图 16-3 DemoCAN 运行图

3．开发板更新固件

DemoCAN 上位机软件开始工作后，按下 TRK-KEA128 板上的 SW3（RESET）按钮，复位单片机。这时单片机中的 Bootloader 程序就开始运行，当发现 CAN 总线上有上位机过来的“FF”后，Bootloader 程序就开始接收和更新用户的应用程序，完成以后如图 16-4 所示。当程序更新完后，就开始转到新的应用程序执行。

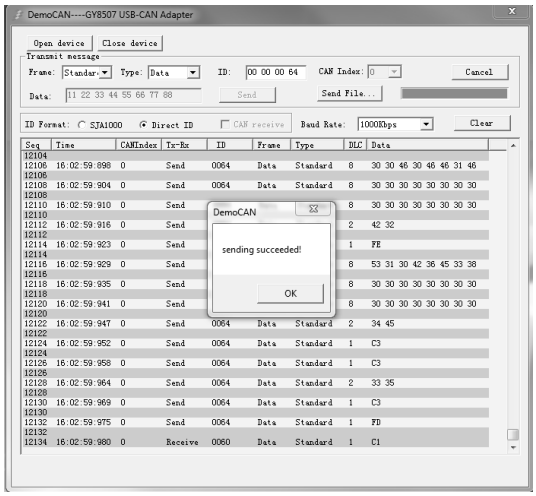


图 16-4　CAN Bootloader 操作成功

16.4.3　地址空间分布

CAN Bootloader 程序放在 0x00000500～0x00003FFF 区域内，应用程序放在 0x000040C0 到 0x0001FFFF 区域内；Bootloader 的中断向量表默认是放在 0x00000000 到 0x000000BF。应用程序的中断向量重定位到 0x00004000 到 0x000040BF 区域内，具体如图 16-5 所示。

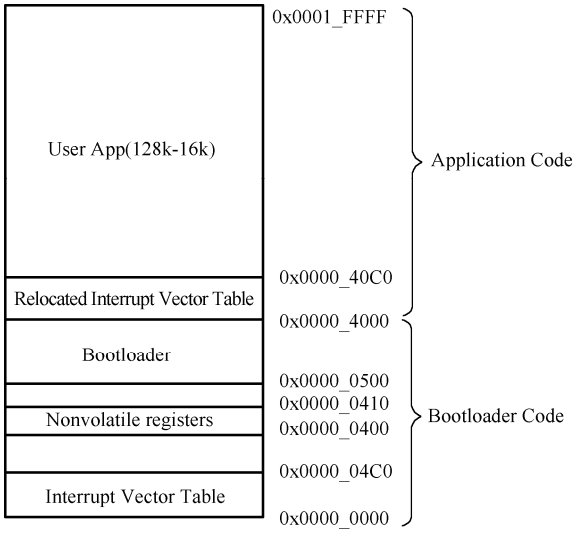


图 16-5　内存分配图

16.4.4　CAN Bootloader 软件流程

上电后，Bootloader 软件首先会查询和上位机握手是否成功；如果握手成功，会接收和更新代码，并最终跳转到新的应用程序的开始代码处执行；如果超时，程序会直接跳到应用程序的启动位置，程序流程如图 16-6 所示。

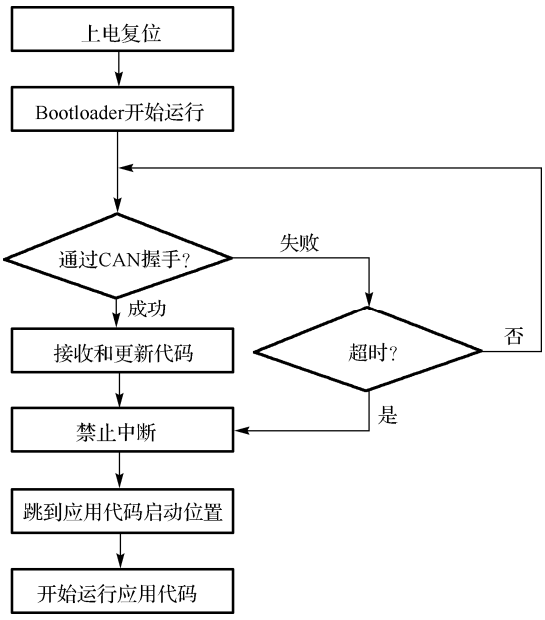


图 16-6 程序流程图

16.5 AUTOSAR 简介

16.5.1 概述

随着汽车电子系统复杂度的不断增长，软件的代码量急剧地上升。目前的行业现状是软件模块化有限，且重用性差；当微处理器型号更换后，软件往往需要重写。大部分嵌入式系统不支持硬件抽象，这给开发带来了很大的困难。AUTOSAR 就是在这样的背景下提出来的。

AUTOSAR 是 AUTomotive Open System ARchitecture（汽车开放系统架构）的首字母缩写，是由全球汽车制造商、部件供应商及其他电子、半导体和软件系统公司联合建立的，该联盟致力于为汽车工业开发一个开放的、标准化的软件架构。飞思卡尔作为加入 AUTOSAR 联盟的第一家半导体厂商，积极地参与 AUTOSAR 系统架构建立的各项活动。AUTOSAR 架构有利于汽车电子系统软件的复用、交换及更新，并为高效管理越来越复杂的汽车电子软件系统提供了一个基础。AUTOSAR 的目标是将汽车系统的基础软件标准化为一个跨 OEM 的“标准栈”，这样就可以集成不同供应商生产的功能模块，也可以使之适应于不同的车辆及不同的车型。

16.5.2 AUTOSAR 软件架构

为了实现 AUTOSAR 的目标，即实现应用程序和基础模块之间的分离，汽车电子软件架构被抽象成几个层，如图 16-7 所示。

基础软件分为以下四个层次。

RTE（Runtime Environment），RTE 层负责 AUTOSAR 软件构件（即应用层）相互间的

通信，以及软件构件与基础软件之间的通信。RTE 层之下的基础软件对于应用层来说是不可见的，必须通过 RTE 进入，它将软件构件从对底层软件和硬件平台的依赖中独立出来，实现了应用程序和基础软件之间的分隔。

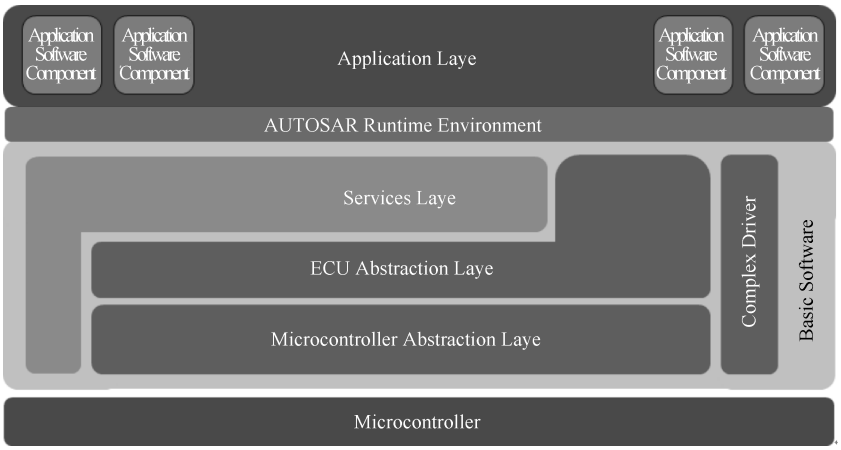


图 16-7　AUTOSAR 软件架构

服务层（Services Layer），该层提供包括诊断协议、存储管理、通信、ECU 模式管理和操作系统等在内的系统服务。

ECU 抽象层（ECU Abstraction Layer），该层将 ECU 结构进行了抽象处理，包含了 ECU 板上外部设备的驱动，以及内部设备与外部设备的接口（I/O），目的是使上层软件与 ECU 硬件设计无关。

微控制器抽象层（Microcontroller Abstraction Layer），该层包括微控制器中内部外设的驱动，以及 MCU 内存映射的外部设备的驱动，目的是为了使上层软件与微处理器型号无关。

除此四层外，为了让某些对时序要求较高的应用模块可以直接访问硬件资源，在 ATUOSAR 软件架构中还定义有复杂驱动（Complex Driver）模块，如喷油量控制和胎压监测等。

16.5.3　飞思卡尔与 AUTOSAR

飞思卡尔作为第一家加入 AUTOSAR 联盟的半导体公司，飞思卡尔可以为 AUTOSAR 的基础软件部分提供 MCAL 和 Operating System。MCAL 驱动主要包括微控制器驱动，存储器驱动，通信部分驱动和 I/O 驱动；同时还可以为存储器硬件抽象部分提供 Flash 模拟 EEPROM 的驱动。

16.5.4　AUTOSAR 相关问题

AUTOSAR 涉及的相关问题非常多，如方法论及标准化应用接口等，由于篇幅的限制未能一一展开，读者如有兴趣可以进一步查找相关资料。可以看到的趋势是，AUTOSAR 正在成为现实，越来越多的 OEM 都计划采用 AUTOSAR。在市场上不少的工具和软件供应商都已经推出了符合 AUTOSAR 标准的工具或者软件，可以为 AUTOSAR 系统的设计和开发提供完成的解决方案。

附录 A

SKEAZ128MLK 引脚功能分配

引脚号		优先级低->优先级高							
80LQFP	引脚名	缺省功能	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6
1	PTD1	DISABLED	PTD1	KBI0_P25	FTM2_CH3	SPI1_MOSI			
2	PTD0	DISABLED	PTD0	KBI0_P24	FTM2_CH2	SPI1_SCK			
3	PTH7	DISABLED	PTH7	KBI1_P31	PWT_IN1				
4	PTH6	DISABLED	PTH6	KBI1_P30					
5	PTH5	DISABLED	PTH5	KBI1_P29					
6	PTE7	DISABLED	PTE7	KBI1_P7	TCLK2		FTM1_CH1	CAN0_TX	
7	PTH2	DISABLED	PTH2	KBI1_P26	BUSOUT		FTM1_CH0	CAN0_RX	
8	VDD	VDD							VDD
9	VDDA	VDDA						VREFH	VDDA
10	VREFH	VREFH							VREFH
11	VREFL	VREFL							VREFL
12	VSS/ VSSA	VSS/VSSA						VSSA	VSS
13	PTB7	EXTAL	PTB7	KBI0_P15	I2C0_SCL				EXTAL
14	PTB6	XTAL	PTB6	KBI0_P14	I2C0_SDA				XTAL
15	PTI4	DISABLED	PTI4		IRQ				
16	PTI1	DISABLED	PTI1		IRQ	UART2_TX			
17	PTI0	DISABLED	PTI0		IRQ	UART2_RX			
18	PTH1	DISABLED	PTH1	KBI1_P25	FTM2_CH1				
19	PTH0	DISABLED	PTH0	KBI1_P24	FTM2_CH0				
20	PTE6	DISABLED	PTE6	KBI1_P6					
21	PTE5	DISABLED	PTE5	KBI1_P5					
22	PTB5	DISABLED	PTB5	KBI0_P13	FTM2_CH5	SPI0_PCS	ACMP1_OUT		
23	PTB4	NMI_b	PTB4	KBI0_P12	FTM2_CH4	SPI0_MISO	ACMP1_IN2	NMI_b	
24	PTC3	ADC0_SE11	PTC3	KBI0_P19	FTM2_CH3		ADC0_SE11		
25	PTC2	ADC0_SE10	PTC2	KBI0_P18	FTM2_CH2		ADC0_SE10		
26	PTD7	DISABLED	PTD7	KBI0_P31	UART2_TX				
27	PTD6	DISABLED	PTD6	KBI0_P30	UART2_RX				
28	PTD5	DISABLED	PTD5	KBI0_P29	PWT_IN0				
29	PTI6	DISABLED	PTI6	IRQ					
30	PTI5	DISABLED	PTI5	IRQ					
31	PTC1	ADC0_SE9	PTC1	KBI0_P17	FTM2_CH1		ADC0_SE9		
32	PTC0	ADC0_SE8	PTC0	KBI0_P16	FTM2_CH0		ADC0_SE8		
33	PTH4	DISABLED	PTH4	KBI1_P28	I2C1_SCL				
34	PTH3	DISABLED	PTH3	KBI1_P27	I2C1_SDA				

续表

引脚号		优先级低->优先级高							
35	PTF7	ADC0_SE15	PTF7	KBI1_P15			ADC0_SE15		
36	PTF6	ADC0_SE14	PTF6	KBI1_P14			ADC0_SE14		
37	PTF5	ADC0_SE13	PTF5	KBI1_P13			ADC0_SE13		
38	PTF4	ADC0_SE12	PTF4	KBI1_P12			ADC0_SE12		
39	PTB3	ADC0_SE7	PTB3	KBI0_P11	SPI0_MOSI	FTM0_CH1	ADC0_SE7		
40	PTB2	ADC0_SE6	PTB2	KBI0_P10	SPI0_SCK	FTM0_CH0	ADC0_SE6		
41	PTB1	ADC0_SE5	PTB1	KBI0_P9	UART0_TX		ADC0_SE5		
42	PTB0	ADC0_SE4	PTB0	KBI0_P8	UART0_RX	PWT_IN1	ADC0_SE4		
43	PTF3	DISABLED	PTF3	KBI1_P11	UART1_TX				
44	PTF2	DISABLED	PTF2	KBI1_P10	UART1_RX				
45	PTA7	ADC0_SE3	PTA7	KBI0_P7	FTM2_FLT2	ACMP1_IN1	ADC0_SE3		
46	PTA6	ADC0_SE2	PTA6	KBI0_P6	FTM2_FLT1	ACMP1_IN0	ADC0_SE2		
47	PTE4	DISABLED	PTE4	KBI1_P4					
48	VSS	VSS							VSS
49	VDD	VDD							VDD
50	PTG7	DISABLED	PTG7	KBI1_P23	FTM2_CH5	SPI1_PCS			
51	PTG6	DISABLED	PTG6	KBI1_P22	FTM2_CH4	SPI1_MISO			
52	PTG5	DISABLED	PTG5	KBI1_P21	FTM2_CH3	SPI1_MOSI			
53	PTG4	DISABLED	PTG4	KBI1_P20	FTM2_CH2	SPI1_SCK			
54	PTF1	DISABLED	PTF1	KBI1_P9	FTM2_CH1				
55	PTF0	DISABLED	PTF0	KBI1_P8	FTM2_CH0				
56	PTD4	DISABLED	PTD4	KBI0_P28					
57	PTD3	DISABLED	PTD3	KBI0_P27	SPI1_PCS				
58	PTD2	DISABLED	PTD2	KBI0_P26	SPI1_MISO				
59	PTA3	DISABLED	PTA3	KBI0_P3	UART0_TX	I2C0_SCL			
60	PTA2	DISABLED	PTA2	KBI0_P2	UART0_RX	I2C0_SDA			
61	PTA1	ADC0_SE1	PTA1	KBI0_P1	FTM0_CH1	I2C0_4WSDAOUT	ACMP0_IN1	ADC0_SE1	
62	PTA0	ADC0_SE0	PTA0	KBI0_P0	FTM0_CH0	I2C0_4WSCLOUT	ACMP0_IN0	ADC0_SE0	
63	PTC7	DISABLED	PTC7	KBI0_P23	UART1_TX			CAN0_TX	
64	PTC6	DISABLED	PTC6	KBI0_P22	UART1_RX			CAN0_RX	
65	PTI3	DISABLED	PTI3	IRQ					
66	PTI2	DISABLED	PTI2	IRQ					
67	PTE3	DISABLED	PTE3	KBI1_P3	SPI0_PCS				
68	PTE2	DISABLED	PTE2	KBI1_P2	SPI0_MISO	PWT_IN0			
69	VSS	VSS							VSS
70	VDD	VDD							VDD
71	PTG3	DISABLED	PTG3	KBI1_P19					
72	PTG2	DISABLED	PTG2	KBI1_P18					
73	PTG1	DISABLED	PTG1	KBI1_P17					
74	PTG0	DISABLED	PTG0	KBI1_P16					
75	PTE1	DISABLED	PTE1	KBI1_P1	SPI0_MOSI		I2C1_SCL		
76	PTE0	DISABLED	PTE0	KBI1_P0	SPI0_SCK	TCLK1	I2C1_SDA		
77	PTC5	DISABLED	PTC5	KBI0_P21		FTM1_CH1		RTC_CLKOUT	
78	PTC4	SWD_CLK	PTC4	KBI0_P20	RTC_CLKOUT	FTM1_CH0	ACMP0_IN2	SWD_CLK	
79	PTA5	RESET_b	PTA5	KBI0_P5	IRQ	TCLK0	RESET_b		
80	PTA4	SWD_DIO	PTA4	KBI0_P4		ACMP0_OUT	SWD_DIO		

附录 B

KEA128 最小系统

模块名称：KEA128最小系统

说明：

1. 芯片SKEAZ128MLK，80FQFP，最大工作电压5 V。
2. C1C10去耦电容，用于滤波，改善系统电磁兼容性，降低电源波动对系统的影响。数量不定，可根据用户实际情况选择。在布PCB板时，分布在MCU四周，且需靠近MCU。
3. KEA128只支持SWD调试接口。
4. KEA128有两种时钟源选择，分别是内部时钟和外部时钟，若使用外部时钟，需要外部晶振电路。

附录 C

printf 格式化输出

C.1 prinf 调用的一般格式

printf 函数是一个标准库函数，它的函数原型在头文件“stdio.h”中。但作为一个特例，不要求在使用 printf 函数之前必须包含 stdio.h 文件。

printf 函数调用的一般形式为：printf(“格式控制字符串”, 输出表列)。

其中格式控制字符串用于指定输出格式。格式控制串可由格式字符串和非格式字符串两种组成。格式字符串是以%开头的字符串，在%后面跟有各种格式字符，以说明输出数据的类型、形式、长度、小数位数等。如：

“%d”表示按十进制整型输出。

“%ld”表示按十进制长整型输出。

“%c”表示按字符型输出等。

非格式字符串原样输出，在显示中起提示作用。输出表列中给出了各个输出项，要求格式字符串和各输出项在数量和类型上应该一一对应。

C.1.1 格式字符串

格式字符串的一般形式为：[标志][输出最小宽度][精度][长度]类型，其中方括号[]中的项为可选项。以下说明各项的意义。

1．类型

类型字符用以表示输出数据的类型，其格式符和意义如表 C-1 所示。

表 C-1 类型字符

格 式 字 符	意　　义
d	以十进制形式输出带符号整数（正数不输出符号）
o	以八进制形式输出无符号整数（不输出前缀 0）
x,X	以十六进制形式输出无符号整数（不输出前缀 0x）
u	以十进制形式输出无符号整数
f	以小数形式输出单、双精度实数
e,E	以指数形式输出单、双精度实数
g,G	以%f 或%e 中较短的输出宽度输出单、双精度实数
c	输出单个字符
s	输出字符串

2．标志

标志字符为–、+、# 和空格四种，其意义如表 C-2 所示。

表 C-2 标志字符

标 志	意 义
–	结果左对齐，右边填空格
+	输出符号（正号或负号）

3．输出最小宽度

用十进制整数来表示输出的最少位数。若实际位数多于定义的宽度，则按实际位数输出，若实际位数少于定义的宽度则补以空格或 0。

4．精度

精度格式符以“.”开头，后跟十进制整数。本项的意义是：如果输出数字，则表示小数的位数；如果输出的是字符，则表示输出字符的个数；若实际位数大于所定义的精度数，则截去超过的部分。

5．长度

长度格式符为 h、l 两种，h 表示按短整型量输出，l 表示按长整型量输出。

C.1.2 输出格式举例

```
#include<stdio.h>
#include<string.h>
int main()
{
    char c, s[20];
    int a=1234;
    float f=3.141592653589;
    double x=0.12345678912345678;
    strcpy(s, "Hello,World");
    c='\x41';
    printf("a=%d\n", a);          //按照十进制整数格式输出，显示 a=1234
    printf("a=%d%%\n", a);        //输出%号结果 a=1234%
    printf("a=%6d\n", a);         //输出 6 位十进制整数左边补空格，显示 a= 1234
    printf("a=%06d\n", a);        //输出 6 位十进制整数左边补 0，显示 a=001234
    printf("a=%2d\n", a);         //a 超过 2 位，按实际输出 a=1234
    printf("a=%-6d\n", a);        //输出 6 位十进制整数右边补空格，显示 a=1234
    printf("f=%f\n", f);          //浮点数有效数字是 7 位，结果 f=3.141593
    printf("f=6.4f\n", f);        //输出 6 列，小数点后 4 位，结果 f=3.1416
    printf("x=%lf\n", x);         //输出长浮点数 x=0.123457
    printf("x=%18.16lf\n", x);    //输出 18 列，小数点后 16 位，x=0.1234567891234567
    printf("c=%c\n", c);          //输出字符 c=A
    printf("c=%x\n", c);          //以十六进制输出字符的 ASCII 码 c=41
    printf("s[]=%s\n", s);        //输出数组字符串 s[]=Hello,World
    printf("s[]=%6.9s\n", s);     //输出最多 9 个字符的字符串 s[]=Hello,Wor
    return 0;
}
```

参 考 文 献

[1] Freescale. KEA128 Sub-Family Data Sheet Rev 4，2014（简称 KEA128 数据手册）.

[2] Freescale. KEA128 Sub-Family Reference ManualRev 2，2014（简称 KEA128 参考手册）.

[3] Freescale.面向汽车行业的 Kinetis EA 系列 MCU（文档编号： KINETISEAMCUFS REV 1），2014（KEA 简介）.

[4] Freescale. CodeWarrior Development Studio for Microcontrollers V10.x Kinetis Assembler Reference Manual，2012（简称 Kinetis 汇编参考手册）.

[5] ARM. Cortex-M0+ Technical Reference Manual Rev.r0p0，2012（简称 M0+参考手册）.

[6] ARM. Cortex-M0+ Devices Generic User Guide，2012（简称 M0+用户指南）.

[7] Free Software Foundation Inc.Using as The gnu Assembler Version2.11.90，2012（简称 GNU 汇编语法）.

[8] Joseph Yiu. The Definitive Guide to the ARM Cortex-M0，Elsevier Inc. 2011（简称 CM0 权威指南）.

[9] Joseph Yiu，著．Cortex-M3 权威指南．宋岩，译．北京：北京航空航天大学出版社，2011.

[10] ARM. ARMv6-M Architecture Reference Manual，2010.

[11] ARM.ARMv7-M Architecture Reference Manual，2010.

[12] NATO Communications and Information Systems Agency. NATO Standard for Development of Reusable Software Components, 1991（简称 NATO）.

[13] 边海龙，贾少华．USB 2.0 设备的设计与开发．北京：人民邮电出版社，2004.

[14] [美]Jack Ganssle, Michael Barr，著．英汉双解嵌入式系统词典．马广云等，译．北京：北京航空航天大学出版社．2006.

[15] [美]Colin Walls，著．嵌入式软件概论．沈建华，译．北京：北京航空航天大学出版社．2007.

[16] [美]Jack Ganssle，著．嵌入式系统设计的艺术（英文版第 2 版）．北京：人民邮电出版社，2009.

[17] 王宜怀，朱仕浪，郭芸，著．嵌入式技术基础与实践——ARM Cortex-M0+ Kinetis L 系列微控制器（第 3 版）．北京：清华大学出版社，2013.

[18] 王宜怀，朱仕浪，姚望舒，著. 嵌入式实时操作系统 MQX 应用开发技术——ARM Cortex-M 微处理器. 北京：电子工业出版社，2014.